CmpE561 Natural Language Processing

Arda Akdemir Burak Suyunu

November 14, 2017

submitted to Tunga GÜNGÖR
Application Project 1
Programming Project

1 Introduction

In this project, we implemented a morphological analyzer for English using the two-level morphology paradigm. The components of the analyzer will have the following properties:

- **Lexicon**: The lexicon includes 100 stems. There are stems with varying lengths; there are several stems of different POSs (nouns, verbs, adjectives, prepositions, etc.); there are also ambiguous stems (e.g. a stem which can be both noun and verb)
- **Morphotactics**: The morphotactics includes all the suffixes used for nouns, verbs, adjectives, and adverbs. Prefixes are out of the scope of the project; only suffixes are considered. It is clear that more than one suffix may be attached to a stem (e.g. compute + - er + -ize + -ation → computerization), and the order of the suffixes are important (e.g. *compute + -ize + -ation + -er). The lexicon includes entries to which these suffixes can be applied..
- **Morphophonemics**: We used e-insertion, e-deletion, y-replacement rules in this project.

2 Input - Lexicon

We have 2 different documents as our dataset.

- **Words.txt**: This document includes stems, POS tag of the stems, and possible initial suffixes that these stems can get. We also included the new form of the stems after getting the suffixes. The format of the document is as follows:

  "stem1" "POS tag of the stem2"
  "stem1+suffix11" "suffix11" "POS tag of the stem1+suffix11"
  ...
  "stem1+suffix1N" "suffix1N" "POS tag of the stem1+suffix1N"
  .
  "stem2" "POS tag of the stem2"
  "stem2+suffix21" "suffix21" "POS tag of the stem2+suffix21"
  ...
  "stem2+suffix2M" "suffix2M" "POS tag of the stem2+suffix2M"

  Example Input:
  compute v
  compute er n
  computation ation n
  computize ize v
  computable able adj
  .
  invest v
  investor or n
  investment ment n
  investing ing n

- **Suffixes.txt**: This document includes suffix interactions which means that which suffixes can a word take that ends with a specific suffix. In each line there are 4 strings:
  "suffix1" "suffix2" "POS tag before suffix2 added" "POS tag after suffix2 added"

  For example; "*al ist n n*" means that a noun word ending with *al* suffix can take *ist* suffix and POS tag of the word changes to noun. Nation*al* + *ist* => Nation*alist*

  Example Input:
  ful ness adj n
  less ness adj n
  less ly adj adv
  al ist n n

al ize n v

3 Program Structure

3.1 Data Structures

We used dictionaries to store our data. We have 7 main variables (dictionaries). (Derived word means stem+affixes)

- **stems**: key: stem word => value: POS tag and list of possible initial suffixes with POS tag of words after adding this suffixes.
  'build': ['v', [['ing', 'n'], ['er', 'n'], ['s', 'v']]] : build is the stem. build is a verb. build can take ing, er, s suffixes and these suffixes convert POS tag of the word build from verb to noun, noun and verb respectively.

- **stems2**: key: derived word => value: [[stem, POS tag]]
  'music': [['music', 'n']] : This is just a compact represantation of the stems to speed up the searches.

- **firstDerivedWords**: key: derived word => value: [[stem, POS tag], [suffix, POS tag before suffix is added, POS tag after suffix is added]]
  'musical': [['music', 'n'], ['al', 'n', 'adj']] : musical is the derived word. music is the stem of the word and it is a noun. al is the suffix music got, and al suffix turns the POS tag of music from noun to adjective.

- **suffixes**: key: suffix => value: [K*[newSuffix, POS tag before newSuffix added, POS tag after newSuffix added]] (K is the number of new suffixes that can be added after the key suffix)
  'ive': [['ity', 'adj', 'n'], ['ness', 'adj', 'n'], ['ly', 'adj', 'adv']] : ive is the suffix. When a word ends with suffix ive which makes the word adjective, can also take ity, ness and ly suffixes and this suffixses change the POS tag of the word from adjective to noun, noun and adverb respectively.
  create + ive + ity = creativity

- **secondDerivedWords**: key: derived word => value: [[stem, POS tag], K*[suffix, POS tag before suffix is added, POS tag after suffix is added]] (K is the number of suffixes that derived word has)
  'receivers': [['receive', 'v'], ['er', 'v', 'n'], ['s', 'n', 'n']] : receivers is the derived word. receive is the stem of the word and it is a verb. er is the first suffix receive got, and er suffix turns the POS tag of receive from verb to noun (receiver). s is the second suffix receive got (or the suffix that reciever got), and s suffix turns the POS tag of receiver from noun to noun (receivers).

- **derivedWords**: This is the union of *firstDerivedWords* and *secondDerivedWords* dictionaries. We use *derivedWords* dictionary and *stems* dictionary to convert words from surface form to lexical form and vice versa.

- **stemsAndDerivedWords**: This is the union of *stems2* and *derivedWords* dictionaries. We use *stemsAndDerivedWords* dictionary for the drawing of FSTs and to speed up surface-lexical form transformation processes. **So with derivedWords, stemsAndDerivedWords dictionaries represent our finite state transducers.**

3.2 Methodology

We create our FSTs in two steps:

1. First, we made a list of stems with their initial suffixes. The word **initial** is important, because we are not intrested in stems' second or third consecutive suffixes. Only the first suffix.

2. Second, we made the Suffixes list which includes the order of suffixes. So, this list actually represents morphotactics. We used this list to generate new derived words by adding appropriate suffixes to the old derived words according to morphotactics.

Important Notes:

- In the first part, we directly created words with one suffixes from stems by hand (firstDerivedWords). Because it is nearly impossible to make an automatical system for 100 words that knows which stem can take which suffix.
- In the second step, we used morphotactics to generate new words on top of the firstDerivedWords, because here we only used suffix-suffix interactions. This word generation process might generate words that are gramatically correct but not commonly used in English.
- While creating new words with morphotactics, we first set a limitation (3) to the number suffix a stem can take. However when we removed the limitation we realized that the number of suffixes a stem takes capped at 5. So, thanks ti our well-designed suffix relations it inherently preserved the natural limitation of suffix numbers.
- We used our ortographic rules while combining suffixes, as mentioned in 2. But, in the first part we didn't use ortograpchic rules. We directly pulled the derived words from the stems lexicon because we wanted the root of the word to be in the correct form. (We could also do it with ortographic rules but because we didn't cover all the rules there were be lots of misspelled words from root).
- As a last note, even if our FST doesn't a real FST, actually it is created in a very similar way. It starts from the stem of the given word and builds up to new forms of this word by using morphotactics and morphophonemics. It stores enough information to show the formation process and to convert structure into both directions (surface => lexicon, lexicon => surface). You can observe both the FST like transition diagram of the word with its stem family and the regular transition in written form.

3.3 Algorithm

1. Read Stems.txt. Create stems and firstDerivedWords dictionaries. Creation of these two dictionaries are very straight forward, because Stems.txt has all the info that is needed.
2. Read Suffixes.txt. Using morphotactics defined in Suffixes.txt create new words on top of firstDerivedWords and store in secondDerivedWords. Use Morphophonemics while combining suffixes.
3. Combine firstDerivedWords and secondDerivedWords dictionaries into derivedWords dictionaries then also combine derivedWords and stems2 dictionaries into stemsAndDerivedWords dictionaries. Those two dictionaries function as our completed FSTs.
4. For a given surface form word, program can transform it into lexical form (and vice versa)
5. Also for a given surface form word, you can see the FST like transition diagram of the word with its stem family.

4 Program Interface and Execution

We used Python 3.4 in Jupyter Notebook environment. You can find the code and necessary documents in our GitHub page. We prepared a very detailed Jupyter Notebook for this project.

To run the program, you need to execute each cell from top to bottom. There are 3 function calls you can use to get outputs: surfaceLex(surfaceFormWord), lexSurface(lexicalFormWord) and drawFSTs(word). In the outputs expressions (e-del), (e-ins) and (y-repl) represent the used orthotactics.While (n->n), (n->v), (adj->adv), etc. represents the POS tag transformation of the word with the addition of the shown suffix.

- **surfaceLex(surfaceFormWord)**: This function call takes a surface form word as a parameter and returns the intermediate and lexical form of the word.
  Example input:
  surfaceLex('methodological')
  Example output:
  Surface Form to Intermediate Form:
  method^logy^ical(y-repl)

  Intermediate Form to Lexical Form:
  method+N+LOGY(n->n)+ICAL(n->adj)

- **lexSurface(lexicalFormWord)**: This function call takes a lexical form word as a parameter and returns the intermediate and surface form of the word.
  Example input:
  lexSurface('method+N+LogY+ical')
  Example output:
  Lexical Form to Intermediate Form:
  method^logy^ical(y-repl)

  Intermediate Form to Surface Form:
  methodological

- drawFSTs(word): This function call prints an FST like transition diagrams that show how program builds up the word from its stem with necessary suffixes. You can observe all the other words from the same stem and also the suffixes and orthotactics.

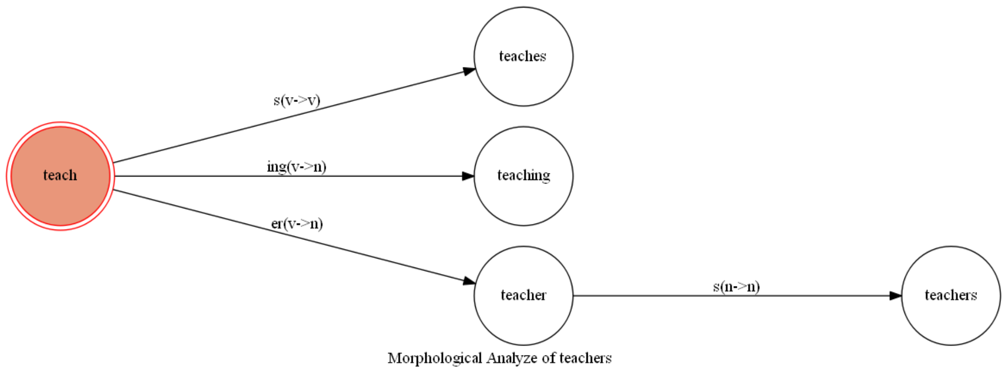

Figure 1: drawFSTs('teachers') - 1. Initial graph. Starting from the stem 'teacher'

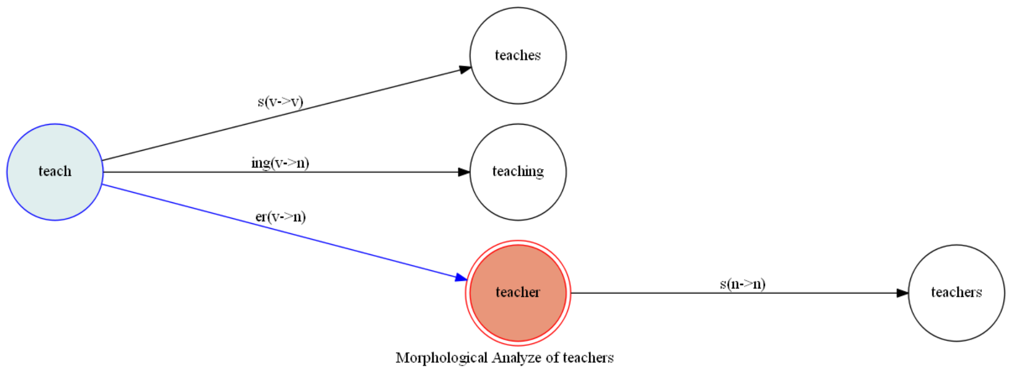

Figure 2: drawFSTs('teachers') - 2. teach + er(v->n) => teacher

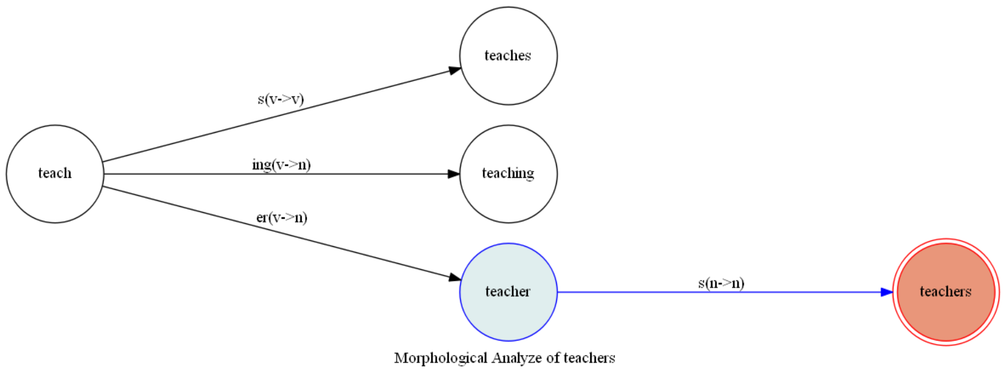

Figure 3: drawFSTs('teachers') - 3. teacher + s(n->n) => teachers

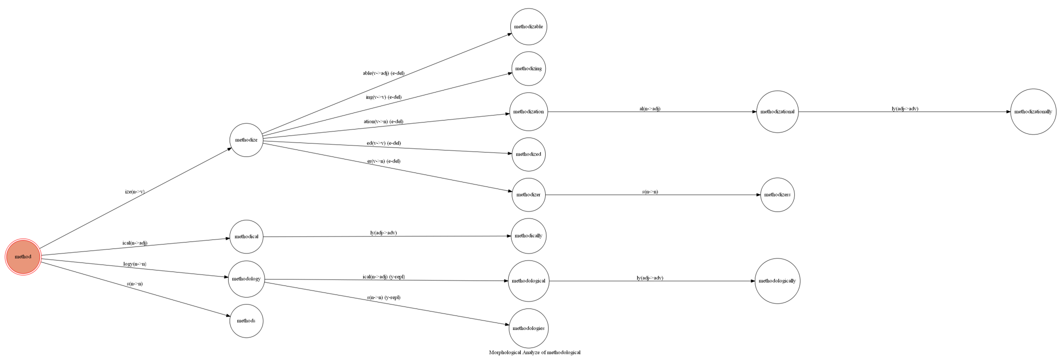

Figure 4: drawFSTs('methodological') - 1. Initial graph. Starting from the stem 'method'

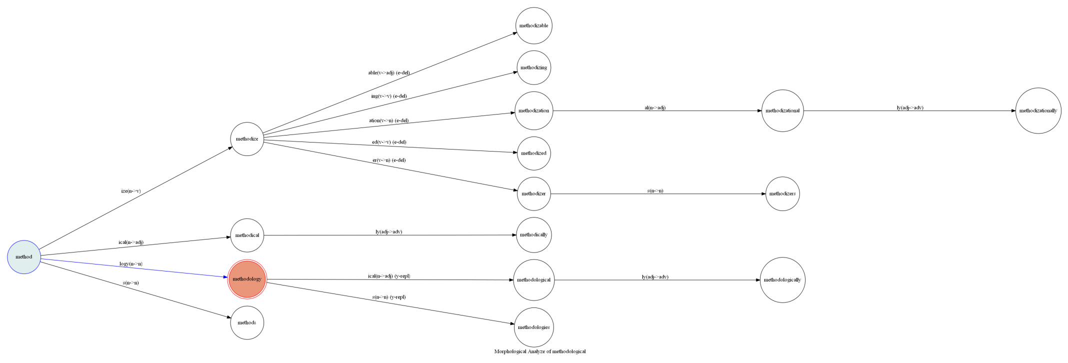

Figure 5: drawFSTs('methodological') - 2. method + logy(n->n) => methodology

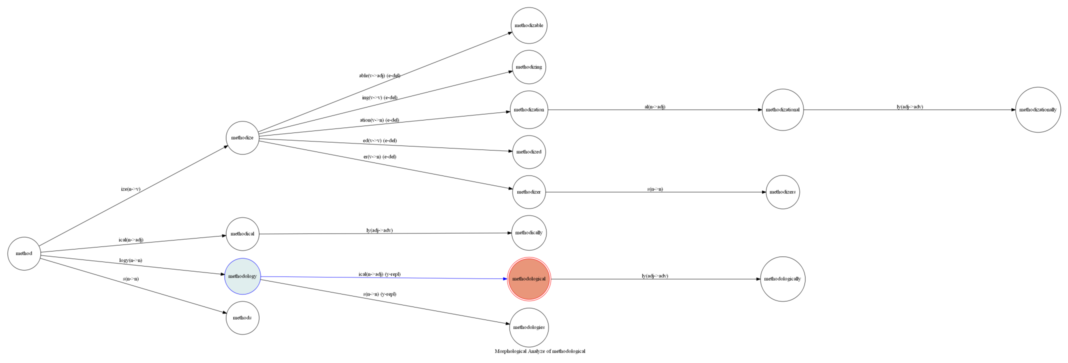

Figure 6: drawFSTs('methodological') - 3. methodology + ical(n->adj) (y-repl) => methodological

5 Examples

It will facilitate understanding the program and the document if some examples regarding the execution are included. For instance, a particular input can be given and the operations and the output of the program under this input may be specified. The examples should be chosen in a manner such that different behaviours of the program may be observed under different inputs.

In this section we would like to explain some of the functionalities of the program by giving some examples. Explaining verbally takes too long and is more difficult to understand thus we include figures of the simplified versions of the FSTs used in our design. They are given as examples and do not cover all the functionality of the program. Further generalization regarding the overall functionality of the program can be made easily after understanding the FSTs shown here.

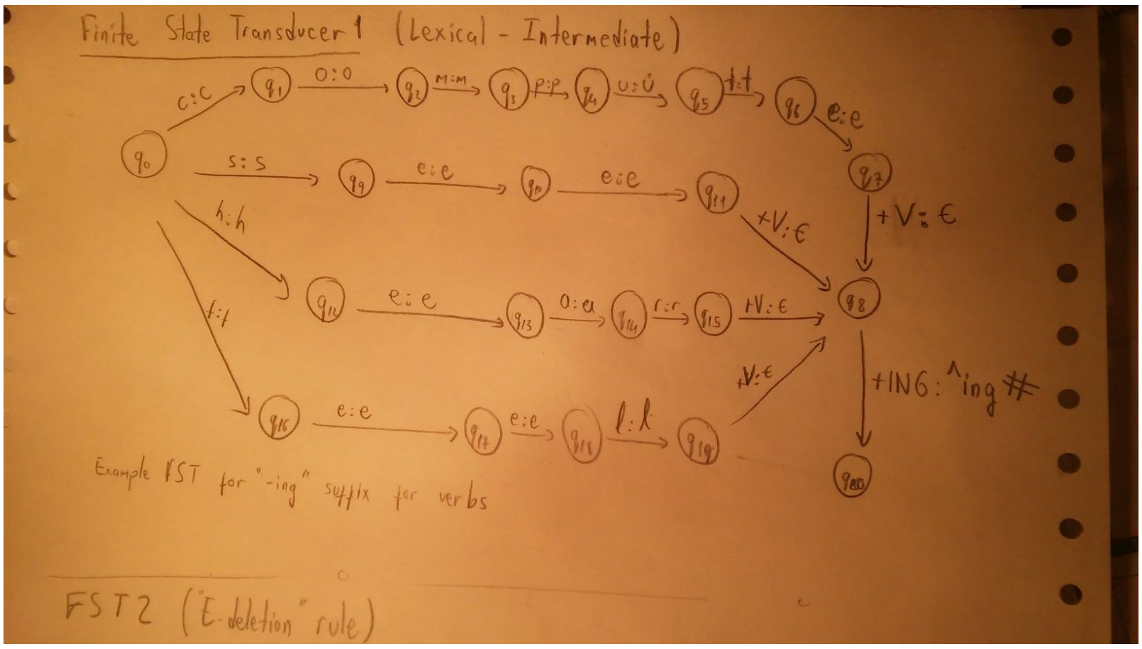

Figure 7: This FST1 shows the workflow of handling Lexical-Intermediate processing of -ING suffix

Figure 1 shows the general workflow between Lexical form and Intermediate form of a given word in the lexicon. The nature of the FSTs allow us to use them bidirectionally by only reversing the input and the output. So this FST can be interpreted in both ways. The transducer simply outputs the given character until reaching the end of the stem which is detected by + symbol. Then according to the suffix attached to the specific word, FST produces different outputs. In this example we only showed the simple example of the suffix -ING but this example can easily be generalized. At this level of FST we only need to append each suffix to output the intermediate form, so no orthographic rules applied to manipulate the characters in the string.

Next example illustrates how multiple suffices are handled in our approach. The morphotactic rules are handled inside this FST. The FST is automatically generated from a list of rules which denotes all the known morphotactics for English. This way we do not enable any words that have wrong suffix ordering such as organizalation etc. As there are no loops in the ordering our system does not generate infinitely long sequences of suffices. In fact the longest suffixation in our design takes 5 suffices. This way we prevent creating a huge lexicon from the given list of stems and suffices. Figure 2 is an example of how order of suffices is handled. In our design we assume that all words that take a certain suffix can all take all the suffices that that suffix can take. This way if lets say a word takes the suffix -al then in our design that word can take the suffix -ly. This assumption makes sense and simplifies the program as the only information required is a list of suffices that can be attached to all suffices.

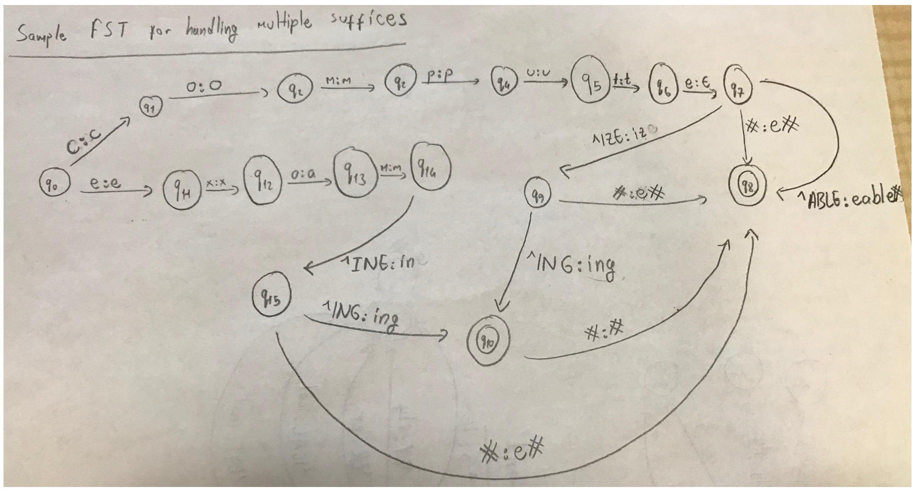

Figure 9: FST3 shows the workflow of handling E-deletion rules. This figure is a simplified version of the overall FST used for handling orthographic rules.

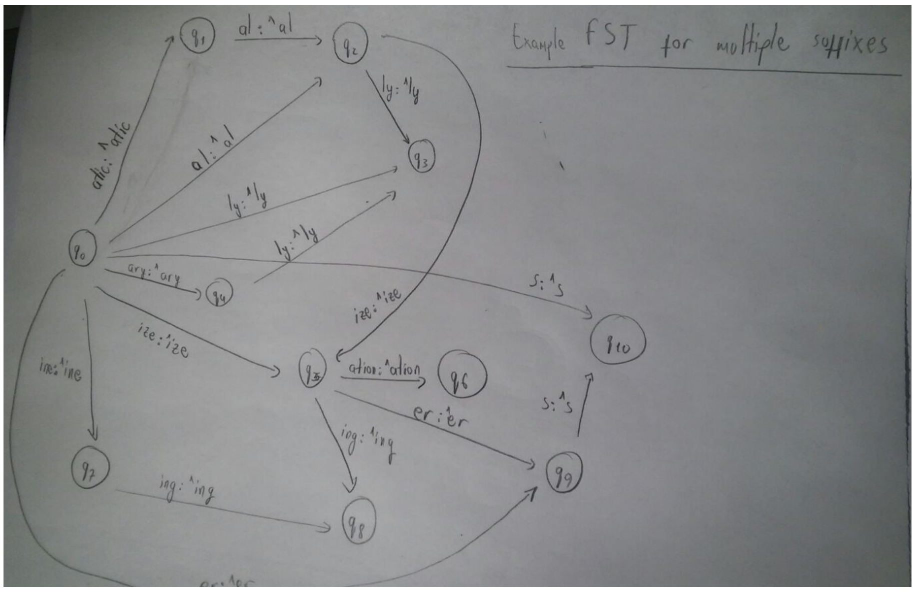

Figure 8: This FST2 shows the workflow of handling multiple suffices. Morphotactic rules are handled at this stage. This is a generalization for suffices. Real FSTs for each word is seperately created in our program

Third figure illustrates how orthographic rules are handled. In our approach, there are manually entered rules for handling three orthographic rules: E-deletion, E-insertion and Y-replacement. The cases in which these rules are applied are manually inputted. The program automatically applies these rules during FST generation thus creating correctly spelled words only. Figure 3 gives a rough idea about how this is done and the idea can be further applied for other rules in the same manner.

6 Improvements and Extensions

In this section, the parts of the program that need improvement and some possible future extensions are discussed. The weak and strong points of the program can be identified. Any shortcomings or defects of the program can be stated. Also, the deviations from the plan at the beginning of the project can be indicated, i.e. the features, user options, data structures, etc. that were initially planned to be included, but could not be done so for some reasons.

Most of the work in this project is spent on manually entering the lexicon and entering the suffices each stem in the lexicon can take. This procedure is feasible, yet takes quiet an effort, for the scope of the project at hand. Yet it is infeasible to try to adapt this approach to a larger lexicon because the combinations become huge as the lexicon gets larger.

We believe that the most important weakness in our design is that we manually entered the stem-suffix relations. As an improvement a Machine Learning approach can be applied to automatically generate the suffices that can be attached to a certain word by learning from a large Corpora. This approach will have mistakes yet it will be robust and requires only a little effort. As the available data in textual form for English is huge, we believe a ML approach is most suitable for automatically detecting word-suffix relations.

In our design we make the assumption that all the words that are attached with the same suffix can be further attached by the same list of suffices. This sounds complicated but can be easily understood with an example. If two words lets say "to plant" and "to compute" can take take the suffix "-ation". Then our design assumes that both words "plantation" and "computation" can further take the suffix "-al". This design enables weird sounding words like "plantationally". Even though they are grammatically correct and obey the orthographic rules such words are not used in English and can be considered as a shortcoming of our design. However we believe that languages are dynamic and subject to change thus such words may be used in the future. Again to overcome this shortcoming, we can use large Corpora to check whether such words like "plantationally" ever occur or not.

7 Difficulties Encountered

The major difficulty encountered in this project is to clearly formulate the task. The majority of the time spent on this project is given to understand what the problem actually is and how to solve it. Once we have successfully understood the task the implementation phase was rather went quiet smoothly.
Another difficulty encountered is the irregular nature of the problem at hand. Languages involve many irregularities and this makes the NLP very challenging. There are many words that do not obey the regular grammar rules and they have to be handled manually. So properly spelling the words and their forms together with suffices require the developers to either find such a lexicon or manually create lexicon. For our project we are required to use a lexicon of size about 100 so it was feasible for us to manually create the lexicon and manually input the irregularities one by one. Yet we also had to consider which words can take which suffixes, because another challenge in NLP is that not all words take all possible suffixes related to that words part of speech. For example not all verbs can take the "-ation" suffix which is appended to verbs. So this challenge must also be handled manually. In our approach we added the a list of suffixes for all words in the lexicon together with how they are combined to form a word. This way we handled the two problems at once. However manually creating this lexicon and checking the word-suffix validities and resulting forms took the major time of the implementation phase.

8 Conclusion

This application project was very useful for us to understand how morphological analysis process works. Furthermore, through this application project we deepened our understanding of FSTs and how they are applied in this area. There are few shortcomings in our design but the size of the lexicon enables our approach to be feasible. If we want to further generalize the application of this approach, we believe making use of ML methods and using large Corpora are the best ways to proceed.