MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
Department of Brain & Cognitive Sciences
6.863/9.611J Natural Language Processing, Spring 2004

Laboratory 1 – Word Parsing
Component II: Warmup Exercises on Word Parsing

Handed out: Feb 9, 2004 **Due: Feb 17, 2004**

Goals of Lab 1, Component II

This laboratory will explore a computational approach to dictionary and morphological analysis – how to "parse" words, and assign them feature and part of speech labels for use in further natural language processing. Component II (this document) is designed for you to gain familiarity with the Athena computers and how to run our system for word parsing, Kimmo. In addition, this lab component is designed to teach you how to think about the two main parts of word parsing: (1) finite-state machines to model morpheme sequences; and (2) finite-state machines to model spelling change rules. The idea behind the Kimmo system is that given this basic two-part machinery, in order to build a dictionary system for a new language 'all' you have to do is write down the descriptions of these two machines – different spelling change rules and different morpheme machines. In particular, we want you to understand how the finite-state machines operate, how rules interact, and how the rules working in parallel model multiple rules. In the last part of this laboratory (handed out on Wednesday), you will do more substantive work, building Kimmo spelling change automata and Kimmo morpheme lexicons to carry out the morphological analysis of a foreign language. (Well, perhaps foreign to a majority of you.)

Reading Preparation: In preparation for understanding the Kimmo system and both parts of this laboratory, you should read the following.

- Component I of this laboratory, which provides background on word parsing and the Kimmo system; Lecture Slides 1 & 2; and Notes 1, all on the course website.
- The Pc-Kimmo website, at http://www.sil.org/pckimmo/ , contains several documents that you should read to understand the system:
  - The general description of the approach: http://www.sil.org/pckimmo/two-level_phon.html
  - http://www.sil.org/pckimmo/v2/doc/rules.html gives in-depth coverage of how the finite-state machines implement spelling-change rules; at a minimum, you should read the section on how the automata actually work, described here: http://www.sil.org/pckimmo/v2/doc/Rules_2.html#subsec:3.2.1
  - The instructions for writing the spelling-change rules at: http://www.sil.org/pckimmo/v2/doc/Rules_4.html
  - A summary of the format for writing rules in section 7 of the online version of Pc-Kimmo that we use, at: www.ai.mit.edu/courses/6.863/pckimmoman.txt
- Optionally, depending on your background:
  - If you are unfamiliar with finite-state machines, read this section: http://www.sil.org/pckimmo/v2/doc/Rules_2.html#subsec:3.2.2
  - If you want to know more about rule conflicts (see also Question 5 below): http://www.sil.org/pckimmo/v2/doc/Rules_3.html#subsec:3.3.11

What you must do and what you must turn in

There are five questions (and an optional bonus question) that you must answer below, in Section 4. We'd like to get by with as little paper as possible for assignments. (Call it ecological necessity.) To this end, we'd like you to write up your answers as a web page for your lab report, and then just email the URL to the TA Catherine Havasi, havasi@mit.edu. Please remember that collaboration is encouraged, but please do write down the names of your collaborators at the beginning of the report. Also, please remember that cloned reports are not acceptable.

2. Running Kimmo: the basics

The Software

We provide two different versions of the Kimmo program, one with an older, command-line interface, `pckimmo`, and one, newly ported to Python, with a graphical interface, `pykimmo`. They both provide nearly the same functionality for generating and recognizing words, tracing a parse, and printing pictures of finite-state machines. The graphical interface has distinct advantages in on-the-fly editing and display. `Pckimmo` has additional tracing functionality and the advantage of long-time stability. There is also a command-line terminal only version of `pykimmo`. The command-line versions of both programs are notably faster than the graphical version, especially with a large lexicon. Either version should give the same results for doing the laboratory, and we will alert you should there be any possibility that this might not be so. Here are the brief instructions for running the two.

- `pykimmo`: login to an Athena workstation and then:

  ```
  athena% add 6.863
  athena% pykimmo
  ```

  This will spit out the commands to use the program in command-line mode if you should want, and then bring up the graphical user interface window, with a toy version of English rules and lexicon already loaded, and a panel into which you can type strings to either generate or recognize (see the picture on the next page). You should load the lab1a rules and lexicon by clicking on the 'load' button and then loading in the rules and lexicon `pykimmo.rul` and `pykimmo.lex` that will show up in a dialog box. You can then proceed with the rest of the exercises below.

- `Pckimmo`: login to an Athena workstation and then:

  ```
  athena% add 6.863
  Athena% cd /mit/6.863/pckimmo-old
  athena% ./pckimmo
  ```

  This will bring up the command line interface:

  ```
  PC-KIMMO TWO-LEVEL PROCESSOR
  Version 1.0.8 (18 February 1992), Copyright 1992 SIL
  Type ? for help
  PC-KIMMO>
  ```

You will now have to load English rules and a lexicon explicitly. You can then proceed with the rest of the exercises.

```
PC-KIMMO>load rules englishpck.rul
Rules being loaded from englishpck.rul
PC-KIMMO>load lexicon englishpck.lex
Lexicon being loaded from englishpck.lex
```

3. Trying out word recognition and word generation

You should now try out generating and recognizing the following strings; check that you get the results we give below. We give the `pckimmo` form for entering strings; in the `pykimmo` graphical version, you just type the strings into the lower left-hand text box and click 'generate' or 'recognize'.

```
PC-KIMMO>generate fox+s
foxes

PC-KIMMO>recognize foxes
`fox+s       [ N(fox)+PL ]
`
PC-KIMMO>generate fly+s
Flies

PC-KIMMO>generate dogg+s
doggs
```

(This example shows that one can generate from any string, without looking at the dictionary. Thus, the best way to debug the rules component is to build it separately from the lexicons – if it does not work, the fault likes in the rules.)

```
PC-KIMMO>recognize spies
`spy+s       [ N(spy)+PL ]
`spy+s       [ V(spy)+3sg.PRES ]
```

You will need to use tracing to see how the machine works to do the exercises. In `pykimmo`, click on 'tracing' and a second window pops up in which you can view the progress of the machine tracking through the rule and morpheme automata. A 'save' button lets you file this for later perusal.

In `pckimmo`, you issue the command:

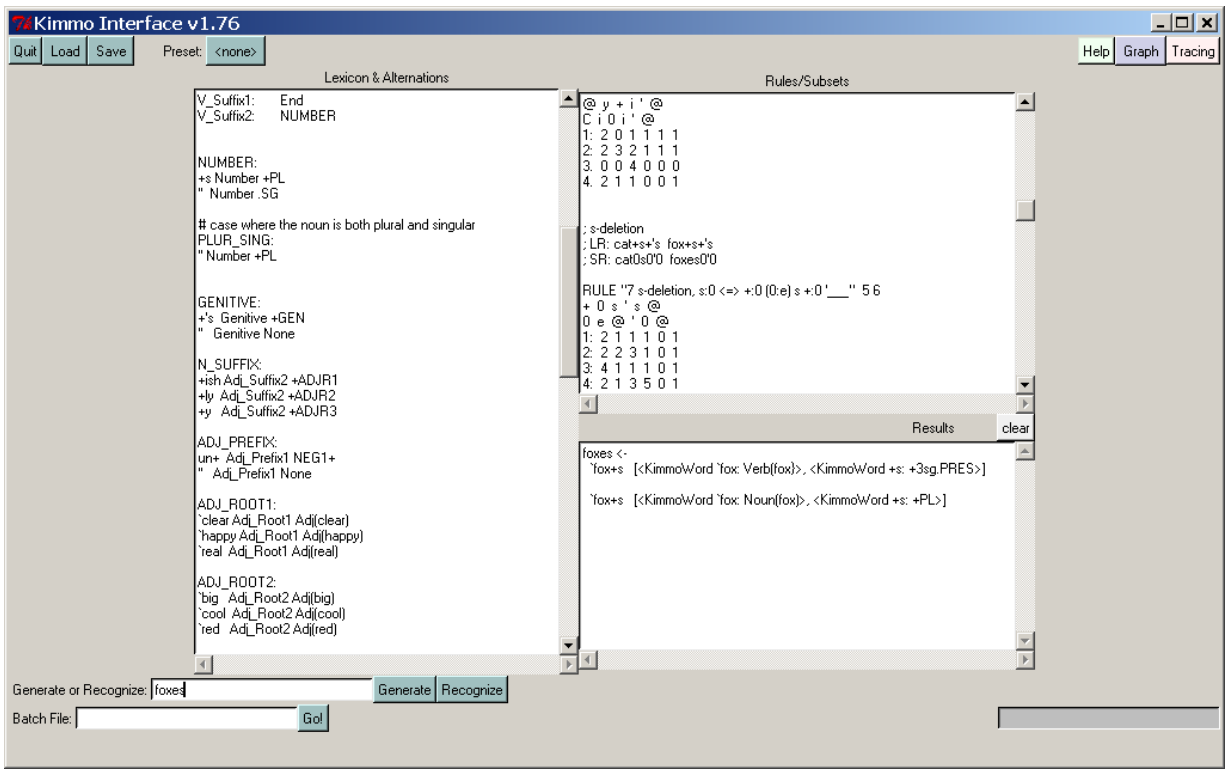
```
PC-KIMMO>set tracing on
```

This information can be captured to a file via the `pckimmo` command `log <filespec>`; and the file is then closed via the command `Close.` When you are ready to get out, in `pckimmo`, you type:

```
PC-KIMMO>quit
```

In the graphical version, you simply click on the 'Quit' button.

Here is a picture of the graphical version.



3. Questions about Kimmo and how it works

For the following questions, please make sure you have loaded either `englishpy.rul` and `englishpy.lex` or `englishpc.rul, englishpc.lex` (depending on which system you run.)

Question 1: Have the system generate from the lexical string refer+ing. The correct result should be 'referring' What is the result that you get? What is going wrong and why? How would you repair it? Test your repair and demonstrate that it works. (Hint: This is not a problem with either the rule or lexicon automata themselves.)

Question 2: Have the system recognize the surface string 'dogs'. What is the result? Note that 'dogs' is also a verb. To accommodate new words and new parts of speech and features for existing words, you have to edit the Alternations/Lexicon file, `englishpyk.lex` or `englishpck.lex`. Please do this so that 'dogs' will also be recognized as a verb – follow the template for other verbs that you see in the `lex` file (please make a copy of the lex file to use in you own directory), and then demonstrate that your addition works. (Note that the lexicon file that `pckimmo` uses is a bit different in format from the one that `pykimmo` uses, but you can accomplish the same result.)

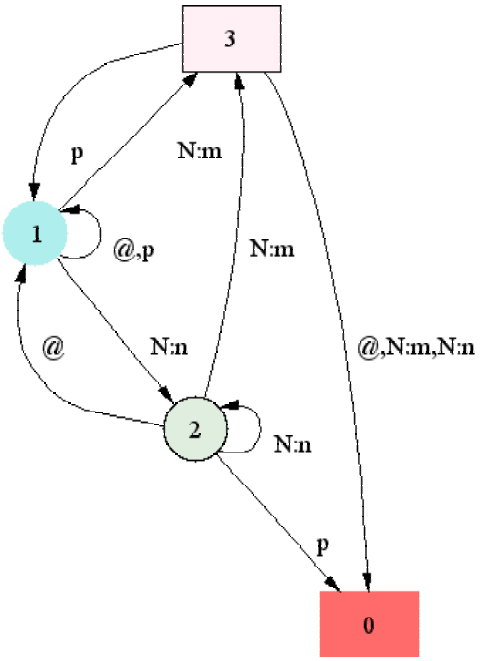
Question 3. Turn tracing on and then have the system recognize the word 'flier'. Examine the trace, and then comment on the following quote from one of the original authors of the Kimmo system: "the recognizer only makes a single left-to-right pass through the string as it homes in on its target in the lexicon." Is this statement accurate? Try to characterize the kind of situation in which the behavior that you observe will arise. Does the same behavior occur with generation? If so, how is generation similar to recognition? If not, how is generation different from recognition?

Question 4: This last question is about the spelling-change rule automata and their interactions. It may be considerably more challenging than the others. Note that it is a question purely about the spelling change automata, and has nothing whatsoever to do with the Lexicon/Alternations. You may want to read the online Kimmo documentation about rule interactions, at http://www.sil.org/pckimmo/v2/doc/Rules_3.html#subsec:3.3.11

Let us hypothesize a language where the lexical morpheme *kaN* is concatenated with the suffix *pat*. For the purposes of this problem, you can ignore all the business about a + morpheme marker being present (it actually doesn't matter for the purposes of this problem). As in many languages, the nasal *N* is 'assimilated' (made phonetically similar to) whatever follows the morpheme boundary – the *p* – by changing it into an *m*. We could write this rule as follows:

Rule 1: N→m | __ p

This says that a lexical *N* is obligatorily replaced by *m* when it appears in the context before *p*. The left context is empty, which means that it doesn't matter here. We can write Rule 1 as the following finite-state transducer (we made this picture by simply typing in the automata description in tabular form, and then having `pykimmo` graph the result). Recall that a singleton arc label like *p* denotes the identity relation, e.g., *p:p*.



Here is the Kimmo tabular format for this automaton:

```
RULE "N realized as m" 3 4
   @ N N p
   @ m n p
1: 1 3 2 1
2: 1 3 2 0
3. 0 0 0 1
```

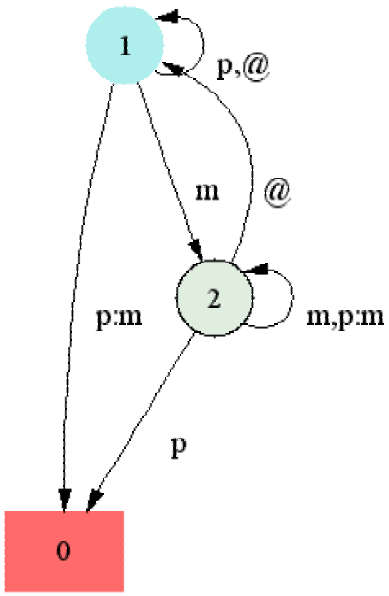
Note how in State 1, the machine waits via looping until it sees an *N:m* pair. Then it goes to State 2. In State 2 if it sees a `p:p` it goes to 1, an accept state, and stays there – with one exception: if it ever sees an `N:n` followed by a `p` , which is barred, it goes to rject state 0. This enforces the constraint that we wanted.

We have placed in the course locker a very short `.rul` file, `rule1.rul,` ready to load into Kimmo to test this rule. Test it out: Please load just this rule into your Kimmo of choice, without a lexicon, and try generating from the lexical form, `kaNpat,` to check that the rule works as advertised. Please note something <u>important</u> about the Alphabet and Default surface character specifications in the first few lines of this rule file. They illustrate that when introduces underlying characters like *N* that will never show up on the surface, one must make adjustments. First, we must of course put *N* in the list of possible alphabetic characters. Second, the default character pairings must be expanded: we must add <u>both</u> *N:m* and *N:n* to the default character list, since either one of these possible pairs could occur by default.

As always though, language is a bit more complicated than just one rule. We further suppose that our imaginary language has a second process that converts an underlying *p* into a surface *m* whenever it occurs after an *m:*

Rule 2: p→ m | m___

The picture for this machine is as follows, then its Kimmo tabular form:



```
RULE "m realized as p"  2 4
      m  p  p  @
      m  m  p  @
1:    2  0  1  1
2:    2  2  0  1
```

Once again, we have placed another very short `.rul` file, `rule2.rul`, in the course locker that represents this automaton. Please load just this rule file in, without a lexicon. Try generating from the form `kampat` to check that the rule works as advertised and produces the output *kammat.* You will note that again we had to add to the default character list to include *p:m* as a possible output pairing.

Now let's put Rules 1 and 2 together. You might immediately see that if these rules are ordered, then they will interact:

- Rule 1 runs first, and then Rule 2.
  - If Rule 1 runs first, then the *N* changes to an *m* (because a *p* follows) and then
  - Rule 2 changes the *p* to an *m*. So *kaNpat* becomes *kammat*
- Rule 2 runs first, then Rule 1.
  - Rule 2 does not apply, since *p* precedes an *N*
  - Rule 1 applies, and changes the N into an *m*
  - The result is *kampat*

Suppose we know that the right surface form is *kammat.* (Suppose that's the one we hear.) The problem is: how can we guarantee that we get the right result in Kimmo? (Remember that all the rules are supposed to apply 'simultaneously'). Try it out. Load the file `rule12.rul` in the course locker, which contains descriptions for *both* of the automata, with Rule 1 coming first. Load this file, and try generating from *kaNpat.* Now change the order of the rules in the file so that Rule 2 comes first, and again try generating from *kaNpat.* Check the results against what we have hypothesized. What is result? Do you have any explanation for this? (Again, remember there is no lexicon involved here – we are only interested in generating the right surface form to pair with an underlying form.)

Since the Kimmo rules are supposed to apply simultaneously, there should be no such ordering effect. What you see is what happens when two rules interact (we say that one 'feeds' the other). To get around this, we have to rewrite the automata in such a way that the correct result obtains. There are two basic ways to revise the Rule 1 and Rule 2 automata to take account of each other and to accomplish this. One approach alters each automaton individually, while the other combines the two into a single automaton. Choose either method, and then write the appropriate Kimmo automata (or automaton) eliminating this ordering problem. Load your revised rule into Kimmo and demonstrate via test output (use tracing) that your revised system works. Depending upon which method you select, comment on your design choice in terms of (1) its possible computational extendibility (how easy would it be to handle, say, 4 or 5 interacting rules); (2) its linguistic transparency (how easy is it to interpret the revised automata in terms of what the natural language does).

Linguists would say that this kind of system has a 'depth' of 2. This is not an unusual phenomenon in natural language. If you would like a challenge (and an early start on a term project), you might consider that the very first modern implementation of a generative

grammar was a morphophonemic system like this that had rule orderings of depth up to 25, with over 400 rules. More astonishing still (at least to my mind), this was all done without the benefit of modern computers, since it was hand-crafted from 1949-1950: it was an account of Modern Hebrew. This was Noam Chomsky's Master's thesis, done at the University of Pennsylvania, *Morphonemics of Modern Hebrew* (1951). A terrific project would be to see whether it is possible to replicate Chomsky's system in Kimmo (it has never been done, as far as I know). Similarly complex cases for English can be found in Chomsky and Halle's classic *Sound Patterns of English* (1968).

Here are the full listings for rule1.rul and rule2.rul

```
; rule1.rul
; Note addition of  N
ALPHABET  a b c d e f g h i j k l m  N n o p q r s t u v w x y z  +
NULL 0
ANY @
BOUNDARY #

;; N:m, N;n needed for rule 1
RULE "surface characters"  1 30
   b c d f g h j k l m    n p q r s t v w x y N N  z a e i o u + #
   b c d f g h j k l m    n p q r s t v w x y n m  z a e i o u 0 #
1: 1 1 1 1 1 1 1 1 1 1    1 1 1 1 1 1 1 1 1 1 1 1  1 1 1 1 1 1 1 1

RULE "N realized as m"  3 4
      N  N  p @
      m  n  p @
1:    3  2  1 1
2:    3  2  0 1
3.    0  0  1 0

END
```

```
;rule2.rul
ALPHABET  a b c d e f g h i j k l m  N n o p q r s t u v w x y z  +
NULL 0
ANY @
BOUNDARY #

;; needed for  rule 2
RULE "surface characters"  1 31
   b c d f g h j k l m p   n p q r s t v w x y N N  z a e i o u + #
   b c d f g h j k l m m   n p q r s t v w x y n m  z a e i o u 0 #
1: 1 1 1 1 1 1 1 1 1 1 1   1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

RULE "m realized as p"  2 4
      m  p  p  @
      m  m  p  @
1:    2  0  1  1
2:    2  2  0  1

END
```