Massachusetts Institute of Technology
6.863J/9.611J, Natural Language Processing, Spring 2004
Department of Electrical Engineering and Computer Science
Department of Brain and Cognitive Sciences
Laboratory 1, Component III: Word Parsing — Building a System

Handed out: February 11, 2004 **Due: February 23, 2004**

1 Introduction

For the second part of Lab 1, you will design Kimmo automata and Kimmo lexicons that can carry out a morphological analysis for a subset of Spanish. The following sections describe the processes that your machines should be designed to cope with. Your lab report should contain three elements as follows. Please email a URL containing your lab report to our TA, `havasi@mit.edu`:

- A description of *how your system operates*. What additional lexical characters have you introduced (if any)? What do your subset definitions represent? What is the purpose of each automaton? Briefly, how does each automaton work? Comment on the extensibility of your system: how difficult would it be to add new lexical items (assuming the same phenomena are covered)? If appropriate, mention problems that arose in the design of your system.
- Pointers to your `.lex` and `.rul` files, so that we may test your files.
- A pointer to a log (batch) record of a recognition run on the test file `spanish.rec`. This file may be found at the top-level of the course locker and also on the web page link for this laboratory. To generate the run, you can use the `batch file` entry in pykimmo, or the `log` command in pckimmo.

Recognizably incomprehensible lab reports will not be graded. If you are a native Spanish speaker, you may find that your judgements disagree with those included here. The ones included here (real bugs aside) are to be considered the "gold standard" to use; we understand that dialect variants may exist. (If you are industrious and clever, you may want to think about how dialectical variation might be accommodated in a system like this one.) By the end of this lab, you should have a very good idea about how to design a word parser for a real language.

Important: These labs have been used before and debugged. However, if you find an error in the laboratory assignment, please let us know (preferably by email to the lab TA and/or our discussion bboard), so that we may inform the rest of the class.

In the following sections we describe the automata (`.rul`) and lexicon (`.lex`) formats, along with some of the (simple) tools we provide to aid in building the automata and visualizing them. After this preliminary, we turn to the three Spanish phenomena we want you to handle with your system.

1.1 Kimmo Automata Format

You are to assume that Spanish orthography uses the following characters, some of which we have introduced to stand for accented characters.

```
a ^ b c C d e < f g h i { j J k l m n ~ o * p q r s t u > v w x y z Z
```

Here, `^` denotes an accented a, (á), `<` denotes an accented e (é); `{` an accented i (í); `*` denotes an accented o (ó); `>` denotes a u (ü) accent; and `~` denotes an n tilde (ñ). For example, `l^piz` stands for *lápiz*. In addition, by convention we use capital letters to denote characters that have a special meaning for the lexical forms. In particular, `C` stands for a possible c softening, `J` stands for possible g softening, and `Z` stands for possible z insertion.

We now describe the general format for rule automata and then for lexicon automata. Essentially, what you are doing here is specifying the states and transitions between states, and the start and final states for these machines. Given this, your automaton (`.rul`) file should be formatted as follows. We assume the format for pykimmo and note differences, if any, with pckimmo where relevant. (It may useful in either case to refer to the online manual, see: `http://www.ai.mit.edu/courses/kimmoman.txt`. Additionally, the turkish rule and automata files that are in the course locker provide valuable guidance as to the rule format.)

1. The very first line of the `pykimmo` file should declare your alphabet characters, the individual tokens that you will use to describe Spanish. Note that we will also declare the special characters +, 0, and # here. If you are using `pykimmo` this may done with a single SUBSET command headed by the special character @; for pckimmo (and alternatively for `pykimmo`) you use the command `ALPHABET`. The order of characters on a line is irrelevant (aside from the @ in SUBSET).

   ```
   ;pykimmo

   SUBSET @ a ^ b c C d e < f g h i { j J k l m n ~ o * p q r s t u > v w x y z Z + 0 #

   ;pckimmo (alternatively, pykimmo)

   ALPHABET a ^ b c C d e < f g h i { j J k l m n ~ o * p q r s t u > v w x y z Z +
   ```

   If you later decide you need additional characters for handling Spanish phenomena, you must modify this alphabet.

2. If you are using pckimmo, next declare your `NULL` (empty, zero), `ANY` (automata relative wildcard), and `BOUNDARY` (end of word) characters. These lines are superfluous in `pykimmo`:

   ```
   NULL 0
   ANY @
   BOUNDARY #
   ```

3. Next, we declare the `SUBSETS` of characters we want to refer to — groups of characters that will be referred to in the automata rules. A `SUBSET`, then, is an abbreviatory device so that you don't have to keep mentioning each individual character by name, when it is really a GROUP of characters that you mean. For example, in English, the characters `s, x, z` are often called *Sibilants* because the Sibilants behave alike in many rules, like the one that inserts an `e` in `fox+s` to yield `foxes`. You should use only the subsets that are necessary for handling the phenomena described in this lab.

   SUBSETS are denoted as follows. For example, in this lab you will require (say) the subset V to stand for the class of vowels (V):

```
SUBSET   V     a e i o u ^ < { * >
```

   You may also find it useful to group vowels according to the following subsets: Back (B), Front, Low, and High (For our purposes, these refer to the way in which the vowel sound is made — e.g., a Front vowel is articulated at the front of the mouth. Try it yourself — that is the great thing about language, you can try all these experiments on yourself, without clearing human subject protocols.)

```
SUBSET BACK  u o a >  * ^
SUBSET FRONT e i < {
SUBSET LOW   e o a < * ^
SUBSET HIGH  i u { >
```

   Finally, you may need to declare a subset for consonants:

```
SUBSET CONSONANTS b c d f g h j k l m n ~ p q r s t v w x y z
```

   Summarizing so far then, the first lines of your `.rul` file should look like this:

```
;  semi-colon or #  is a comment, like Lisp - put them anywhere
;  This is the format for the .rul file for Spanish. Note that blank lines are ignored

;   + = morpheme break
; For pykimmo
SUBSET @ a ^ b c C d e < f g h i { j J k l m n ~ o * p q r s t u > v w x y z Z + 0 #
SUBSET   V     a e i o u ^ < { * >
SUBSET BACK  u o a >  * ^
SUBSET FRONT e i < {
SUBSET LOW   e o a < * ^
SUBSET HIGH  i u { >
SUBSET CONSONANTS b c d f g h j k l m n ~ p q r s t v w x y z
```

4. At this point, you skip one or more blank lines and then are ready to specify the rule automta (transducers) themselves. Your very first automaton should define the *default* `lexical:surface` pairs. This is required as an 'otherwise' condition: because the automata rules themselves block various possible pairs from appearing, we need some escape clause to ensure that there is *some* way for, say, an `c:c` to 'surface'. This automaton will always have just one state, which is both a start state and an accepting state, and transitions from this state to itself, labeled with all the possible default `lexical:surface` pairs.`fst` Thus, if there are 33 pairs, the resulting automaton table will have one row and 33 columns. (Note that purely underlying characters like C and Z that **never**

appear on the surface need *not* be included; however, the pairings like `+:0` and `@:@` typically are always required. In addition, in `pykimmo`, `#:#` is also required.)

You can specify this default automaton in one of three ways:

- By writing down the automaton table explicitly (see the first few lines of the `turkish.rul` file for an example). Note that you can break this automaton into more than one part if you want.
- In `pykimmo`, by using the `Default` statement to simply list all the characters;
- By using the program `fst` that will automatically process a set of rules and write out the corresponding tabular representation, including the default automaton.

5. Follow the default automaton with the actual set of automata that you write for Spanish. **This is the beginning of the actual work you will do for the laboratory — your contribution to the `.rul` file.** (The names below are just made up of course — the names and the number of automata is up to you.)

```
RULE "pluralize"
.
(automaton table)
.
RULE "ctoz"
.
(automaton table)
.
```

6. The very last line in your rule file should be:

```
END
```

Once again, **note** that if you use the program `fst` you NEED NOT construct these tables as given above. You **will** have to describe the rules to `fst` in a form as described below. It is worth mentioning that the older program `pckimmo` currently does much more error checking for the syntax of rule files, and provides better error messages — so one strategy is to use that program to do very quick syntax checking.

If you desire more insight into what a `.rul` file should look like, please examine the file for english, `englex.rul` or `turkish.rul` in the course locker in the `pykimmo` subdirectory.

1.1.1 Building automata using `fst`

Writing out the automata by hand is something only computers, not humans, should do. "Compiling" a set of rules written in so-called "re-write rule" format, e.g., "i goes to e except after c" into correct automata is in general quite tricky to work out. As you saw in Lab 1, Part 1, This is due to ordering conflicts between general and more specific rules that refer to the same context. As mentioned, writing such a compiler would make an excellent term project.

To make the process of writing the rule automata a little bit easier easier, we have supplied a very simple rule "re-write engine", called `fst`, that takes over some of this drudgery. We also suggest using the graphing component of `pykimmo` to view and print transition diagrams for the automata you design — this make debugging much easier.

How do you use fst and what does it do? Using it is simple. After you've added 6.863, you simply cd to the course locker, `/mit/6.863/` and then:

```
athena% fst -o outfile infile  e.g.,
athena% fst -o <your dir path>/turkish.rul <your dir path>/turkish.fst
```

where `infile` is a much simpler description of the automata and `outfile` is the tabular representation required by Kimmo. Fst is far from perfect, and sometimes requires some 'post-editing' of the tables to ensure that they work correctly. This is especially true if you use fst to produce input to `pykimmo`. So, please look over the output of this program — run the tables through `pykimmo` to get pictures of the automata. You have been warned! Still, in the past, people have had good success using `fst`. Whatever you decide to do, remember that in the end you are to turn in the resulting automata rules themselves in the (the `.rul` file), though it is helpful to supply the `fst` input as explanation.

Here is an example of what an `fst` file looks like. This also gives us an opportunity to again briefly describe the tabular format for automata.

```
    subset vowel a e

    machine "ken"
    state foo
    vowel:vowel bar
    b:b foo
    c:c foo
    d:d foo
    others reject

    rejecting state bar
    b:e foo
    b:b reject
    others foo
```

This machine implements the well-known grammar rule, "b after a vowel turns to e." It contains two states (called `foo` and `bar`), the first is the starting state. If it sees a vowel paired with another vowel, it goes to state `bar` where, if it sees a `b`, it must be paired with an `e`. The state table generated is smaller than what fst uses, but harder to edit (note how the rejecting state is automatically accommodated in fst).

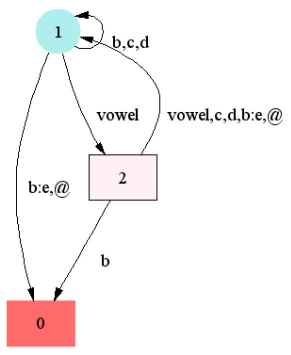
```
ALPHABET a e b c d
NULL 0
ANY @
BOUNDARY #
SUBSET vowel a e

RULE "Default character pairs" 1 6
 a e b c d @
 a e b c d @
1:  1 1 1 1 1 1

RULE "ken"   2 6
     vowel b c d b @
     vowel b c d e @
  1:     2 1 1 1 0 0
  2.     1 0 1 1 1 1
END
```

The format for automata specifies the possible transition pairs along the top row, in columns, followed by the states, one per row. Next state transitions are indicated by the number in the table, indexed by the (row state number, column transition label) pair. The start state is always the first row, state 1. Final states are marked with colons; nonfinal states with a period. The rejecting state 0 need not be listed in a row. The number of rows and columns is specified in the first line of the rule, along with a mnemonic name in double quotes.

This is the picture produced by `pykimmo` corresponding to the `ken automaton`:



Things to note: `fst` creates the alphabet for you, and makes assumptions about the `null`, `any`, and `boundary` characters. To get letters into the alphabet that you never otherwise use, put them in another subset. Note that fst creates the default pairings for you, so that Kimmo knows, e.g., that `x` can always pair with `x`.

Not unexpectedly, `fst` has limits: self-referential rules cause infinite looping! (They are also wrong to write in the first place, however.)

Having discussed the automata format, we next turn to the Lexicon automaton— the specification of the fsa that describes the possible morpheme combinations in a language.

1.2 Kimmo Lexicon Format

The morpheme finite state automaton is written in what the system calls a `.lex` file. `pckimmo` and `englishpck.lex` have slightly baroque, and slightly different ways of specifying this fsa. Recall that what we must do is provide the states of the fsa and their possible transitions, where the states are morpheme classes and the transitions are (usually) string sequences that move the automaton to a possible successor morpheme states. The `.rul` file does this using two distinct section formats: one to describe morpheme states and their possible sets of transition labels; and a second section to explicitly spell out the transition labels between states — what input tokens (as transduced by the spelling machines!) take us from one state to the next.

The automata descripton schema looks like the following (for details on pckimmo, see the online manual section in `http://www.ai.mit.edu/courses/6.863/kimmoman.txt`, page 20):

```
;;State Names        Transition Labels

Begin:               Transition-label-set1  ....  ...     End
State name-1:        Transition-label-set1  Transition-label-set2 ...   Transition-label-setn
...
State name-n:  ... ...

; conclusion of state names and transition label sets;
; now the format for the labels and next-states

Transition-label-set1:
transition label 1:           Next-state        Output symbol as result of transition
transition label 2:           Next-state        Output symbol as result of transition

;;..more transition labels...

Transition-label-set2:
...

End:
'#'    Begin   None
```

From this one can see that there is always a distinguished `Begin` state, with one more transition label sets, and concluding with a possible `End` transition on the word boundary `#` back to the `Begin` state again (so as to be able to process more words) — this is also an accepting state if there is no more input.

All states of the machine are given on the left-hand side of a line, followed by possible transition label sets (equivalence classes of character sequences) that can exit from this state. As an example, consider the `englishpyk.lex` file. It starts as follows:

```
Begin:         N_ROOT  ADJ_PREFIX  V_PREFIX  End
N_Root1:       N_SUFFIX NUMBER
N_Root2:
N_Root3
N_Suffix:
Number:
Genitive:        End
...
V_Suffix1        End
V_Suffix2        NUMBER
```

In this English example, the `Begin` state has four possible exit arcs, labeled with the sets of tokens specified by `N_ROOT`, `ADJ_PREFIX`, `V_PREFIX`, and `End`. It is important to remember that the labels on the righthand side are *not* state names at all — they are sets of transition labels. This can be especially confusing to beginners because (as in this case) there is a `Number` state that has the same name as the set of `NUMBER` transition labels. (Here we are following the original pckimmo design —almost certainly not good practice, since the names can be arbitrary.) In any case, it is easy to avoid this confusion: if you draw the lexicon fsa with pykimmo, we have colored the morpheme states cyan, and the transition label sets white. So please use this handy tool; see the picture below.

After all the states are specified, one provides an expansion of the transition label sets . We do this by first the transition label name followed by a colon; then, on a new line, listing all the individual elements of that transition label class, followed by the next state that transition leads to (i.e., the name of some lefthand side morpheme state), and then, finally, the output that the machine should produce given that it makes this transition. Again, to take an example from the `englishpyk.lex` file, we show the expansion of the transitions for `N_ROOT`:

```
N_ROOT:
cat       N_Root1    Noun(cat)
dog     N_Root1    Noun(dog)
...
person  N_Root2   Noun(person)
...
sheep   N_Root 3   Noun(sheep)
```

Here we see that the set of transition labels for `N_Root` is expanded as the list cat, dog, ..., person. Thus, `cat` is one of the actual character strings required to make a transition from the state `Begin` to the state `N_Root1`. If the machine does so (i.e., it actually finds this string, as checked by the spelling change pair machines), then it outputs the string `Noun(cat)`, which we may think of as a gloss for the root word. (It could be anything of course — for example, the translation of the word into another language, like Italian. You will be able to use this when you do Spanish.)

In the pykimmo picture of the corresponding fsa, we do not have space to explicitly write out all these arc labels between states, so we simply draw them as their transition label set names, in white.

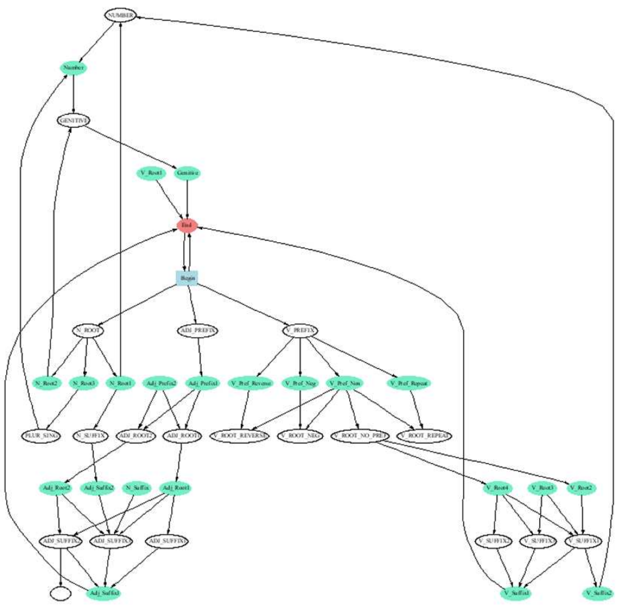
The last transition label description is the one for the special `End` transition, that leads us to an accepting state. In this case, we make the `Begin` state the accepting state, and the machine gets there by finding an end-of-word marker `#` in the input. The output is the special empty element `None` — i.e., nothing is output. This always concludes the construction of a `.lex` file:

```
End:
'#'     Begin    None
```

It is up to the automaton writer to ensure that there are no "dead states" or transitions that lead nowhere, as well as other cycles in this system that are inappropriate (besides the one that goes back to the start state to accept). (You might recall that we might actually require loops to capture words such as *antianti...missile*. It is certainly true that one could easily add automatic checks for these degenerate conditions.) Part of your job will be to construct this fsa and then use the graph tools to debug it. You have to figure out the right morpheme classes and transitions on your own — not always a simple task. Again, for reference, you should consult the sample lexicon files in the pykimmo or pckimmo directory.

To repeat earlier advice, remember that the lexicon file does *not* concern itself with spelling changes — that is left to the `.rul` file. The lexicon must be used if one is recognizing words (because it is the only component that knows what possible roots and affixes there are). Generation can proceed without a lexicon.

(To see the fs machine for the lexicon, it's best to use pykimmo directly.)



Now we turn to the actual Spanish phenomena for which you are to build your rules and lexicon.

1.3 Morphological Phenomena in Spanish

In this lab there are three basic Spanish word formation phenomena that we want you to handle, described below. You should demonstrate the operation of your system on *all* of the Spanish forms that are listed as examples in the following sections. There is a complete list of all the words your final system should handle in the file `spanish.rec` in the course locker. Here is what we mean by "correctly handle": there are 47 "good" forms that your system should recognize as words and correctly parse, and 13 "bad" forms that your system should reject (i.e., output "None"). You may wonder why we do not provide a generator test file. That is because if we provided that file it would immediately tell you how to write the lexicon morpheme fsa — no challenge there.

It is suggested that you debug your automata (`.rul` file) before trying to create a dictionary (`.lex` file). This is because you can debug the rules without a lexicon via the process of trying to generate surface forms, while recognition requires both rules *and* a working lexicon. In addition, it may be useful to construct your automata by a process of successive approximation, bringing in new phenomena after you can handle old ones. It is especially important to remember that the automata act as *constraint filters*, ruling out impossible surface/lexical possibilities, and letting through all the remaining possibilities.

Hint 1: Certain of the solutions (e.g., adding inflections to root forms) will involve programming the automata in a way that appropriately handles the phenomena. Other cases (e.g., internal root changes) will involve judicious choices of underlying forms in the lexicon.

Note 1: It is important that your final system neither under-generate nor over-generate. That is, it should *not* accept invalid Spanish word forms (over-generation), nor fail to parse valid ones (of those that we provide). Your finished product must be able to recognize and generate **all and only** the forms in the file `spanish.rec`, which is in the course locker.

Further, your finished system should generate exactly **one** output form when recognizing a word, except in the case of true ambiguities (as with *flies* in English, which is both a Noun and a Verb). Finally, it is a bad sign if your system generates multiple copies of the same form This is not only an indication of something amiss in the design, but it is a computational burden: a system that generates 5 copies of "pagamos" , say, would be hard to reckon with in a post-Kimmo processing phase.

Note 2: We do understand that some of you might not be native speakers of this very beautiful language. Since some of this laboratory involves conjugating Spanish verbs, for reference we provide a link to a Spanish verb conjugator (in case you want to know how *decir* comes out, for example:

`http://www.verba.org/`

On the other hand, as we mentioned at the outset, if you are a native Spanish speaker, you might have dialect variations that don't agree with what we state here — that's OK, but you should take what is written here as the standard. (How does one pronounce the name of the city "Barcelona" for example? See below.)

Note 3. THIS IS VERY IMPORTANT: **NONE** of the Spanish words and word forms that you see below are meant to be the *lexical* or underlying forms that the word parser should recover when recognizing a word. Rather, they are the *surface* forms (roughly, what we hear). Please DO NOT mistake them for lexical forms even though SOME of them may look like that because they have the same character sequences — e.g., *e r* looks like *e r*. But this is just a case where the lexical and surface characters happen to be identical. The whole point of Kimmo is to figure out the mapping between lexical and surface forms — and you are to figure out that mapping yourself. That's what makes it an interesting exercise.

What appears in your lexicon are the stems or root forms — underlying forms — of what is on the surface. This is not so. In particular, do NOT assume that, for example, the surface form *coger* IS the underlying, root of this Spanish verb. Yes: this might have been what you learned in grammar school, but you must throw away this notion. Nor is the root identical with the 'infinitive' form that you memorized when learning Spanish or French verbs (or whatever). Rather, the infinitival form of this verb, *coger* is **itself** derived from the real underlying lexical form, plus the suffix, i.e., `+er`. I will say it one more time: you CANNOT just take the infinitival forms that you might be familiar with (or might be listed below) and say that these are the underlying, lexical forms (though some might be identical looking character strings in some cases). Your goal is to arrive at the most general root plus affix system possible — the most compact. You cannot just list out individually all the cases. That defeats the purpose.

Turning now to some substance, here are the three phenomena we'd like you to handle: (1) g-j-mutation; (2) z-c-mutation; and (3) plural formation. We describe these next in turn.

1.3.1 g-j mutation

The first phenomenon that your machine should handle is known as *g-j mutation*. The canonical example for this phenomenon is the verb *coger* (catch, seize, grab), where the consonant *g* becomes *j* before back vowels, but *g* otherwise:

```
coger (infinitive)
cojo (pres indic 1p sg)
coges (pres indic 2p sg)
coge (pres indic 3p sg)
cogemos (pres indic 1p pl)
cogen (pres indic 3p pl)
coja (pres subj 3p 1p sg)
```

There are other verbs, however, that are not subject to this rule:

```
llegar (arrive)
llego
llegan
pagar (pay)
pago
pagan
```

You are to accept all of the verbs above, but not the ill-formed words:

```
llejo
lleja
cogo
coga
```

Hint 2: You could use the lexical character `J` to solve this. (But there are other solutions that do not require a `J`.) You may also want to define special characters like J and Z for the *o*→*ue* vowel changes that occur, but you should talk to me first about this, since *real* Spanish vowel changes depend on stress in a complicated way that we do not describe here.

This issue of irregular verbs (decir, cocer) in Spanish and how to deal with them in the Lexicon component versus in the Rules component is an issue with real bearing on the study of the human language faculty. As you know, irregular verbs in English "spring-sprung, sing-sung, dig-dug" *could* be handled in the Lexicon on a case-by-case basis (in fact, this sort of ad-hoc solution has recently been proposed by S. Pinker in his next-to-latest bestselling airport novel). Of course, a more insightful generalization might note that nasals followed by coronal stops are hard for humans to articulate, like "digd," and that there is a systematic change from `i`→*u* that occurs for these forms. Thus, whether irregular past tense verbs are hard-coded in the lexicon or produced in a generative component is an issue of serious debate in the cognitive sciences these days, and as people who actually deal with the implementations of these issues, you may have more to say one day about design, complexity, and redundancy than some of these armchair "the-brain is-a-computer" polemicists. (I can refer interested parties to some interesting work on the acquisition of irregular verbs.)

1.3.2 z-c mutation

For your next Spanish effect to model, note that the consonant *z* becomes *c* before front vowels, but *z* otherwise. The verb *cruzar* (cross) is a canonical example of the application of this rule:

```
cruzar (infinitive)
cruzo (pres indic 1p sg)
cruzas (pres indic 2p sg)
cruza (pres indic 3p sg)
cruzamos (pres indic 1p pl)
cruzan (pres indic 3p pl)
cruce (pres subj 3p 1p sg)
```

If adding an 's' causes a front vowel (e.g., 'e') to surface (see the next subsection) the rule is still applicable. The canonical example of this is the noun *l*ˆ *piz* (pencil):

```
l^piz (pencil)
l^pices (pencil, plural)
```

You are to accept all of the words above, but not:

```
cruco
cruca
crucan
cruze
l^pizes
```

Note 3: I remind you that what we do in 6.863 and in the enterprise of NLP generally is deal with *text*/spelling/orthography, and not pronunciation. Thus, some of you have asked whether the `z-->c` mutation occurs for all words (as opposed to lexically, as in the `o-->ue` dipthongization), and how it is pronounced. In fact, in Castillian Spanish, both 'z' and 'c' are pronounced as *th* (as visitors to Gaudi's *Barthelona* may know) (although with a voicing contrast, as distinguished by distinct phonemes as in English *the* vs. *they*). We abstract away from such variation, since Spanish is uniformly *written* with 'z' and 'c' in these cases, and since no morphological analysis is plausibly carried out with the precision of the International Phonetic Alphabet — there is no reason to change all words with silent p, like psycholinguistics, into a more "true" form. In short, assume for this lab that the `z-->c` rule occurs uniformly (before back vowels) for everything a *text* analysis will ever see.

Pluralization

Adding 's' to a Spanish noun that ends in a consonant induces a surface 'e' to appear. Consider the noun *ciudad* (city):

```
ciudad
ciudades
```

Note that this rule may interact with other rules, e.g., the *z-c-mutation* rule above. This is most apparent for a noun like *l^ piz* (pencil):

```
l^piz (pencil)
l^pices (pencil, plural)
```

Nouns ending in a vowel are **not** subject to this rule:

```
bota
botas
```

You are to accept all of the words above, but reject as ill-formed:

```
ciudads
l^pizs
l^pics
botaes
```

1.4 The Lexicon (Dictionary)

This section describes the suffixes that you should list in your lexicon (dictionary), combining with either nouns or verbs.

A morphological analyzer is intended to be the “front end” for a parsing system, and therefore your dictionary should be designed so that your system recovers syntactic features that would be useful in parsing. For example, the result of recognizing the Spanish verb *venzo* should include a feature indicating that it is a verb with the features first person (1p), singular (sg) and present indicative (pres indic).

In an actual system for understanding Spanish, dictionary entries would also include semantic features of some kind, and as a standin for semantic information, you may wish to include the English translation of a word as a kind of pseudofeature. For example, the result of recognizing *venzo* might also include the ‘gloss’ (conquer, defeat). It may seem that you are being asked to handle a large number of suffixes in this section, but once you have set up the basic structure of your dictionary—in what sublexicon do noun endings go, and so forth—it is much easier to add a new suffix than to design a new automaton. Remember too, that a fullscale morphological analyzer would need a full set of tenses and endings, but you will only consider the present tense (indicative and subjunctive) and infinitival forms. You need not consider other tenses, moods, or aspects.

We will consider only three items here that you must deal with: (1) Noun endings; (2) Verb endings; and (3) some additional, miscellaneous examples (additional verbs).

1.4.1 Noun Endings

The only noun ending you will need is the plural suffix `+s`.

1.4.2 Verb Endings

In general, a verb consists of a verb stem (stored in your main lexicon, `spanish.lex`) and tense endings. The simplest “tense marker” is the infinitive marker, which may be of three types: `+ar`, `+er`, `+ir`. The present tense indicative for each of these three verb types is specified as follows:

Present Indicative			
Person	**+ar verbs**	**+er verbs**	**+ir verbs**
1p, sg	+o	+o	+o
2p, sg	+as	+es	+es
3p, sg	+a	+e	+e
1p, pl	+amos	+emos	+imos
3p, pl	+an	+en	+en

The present tense subjunctive for each of these three verb types is specified as follows:

Present Subjunctive			
Person	**+ar verbs**	**+er verbs**	**+ir verbs**
1p, sg	+e	+a	+a
2p, sg	+es	+as	+as
3p, sg	+e	+a	+a
1p, pl	+emos	+amos	+amos
3p, pl	+en	+an	+an

For every verb listed previously, and for the verbs listed in the next section, we want your system to be able to parse the proper conjugation, yielding the root form plus the right ending. For example, `cojas` has an underlying root form of and must be parsed as `coja+as` (catch, 2p, sg). (Note how one `a` gets erased.)

1.4.3 Examples

Here are a few more examples of subjunctive verb forms (not previously listed) that your system should handle., where handle means "correctly parse and return the proper gloss for". You may note that they are verbs, so should follow the conjugation tables given above.

Remember, you can grab the complete list from the website under the "resources" section, as `spanish.rec`.

```
cojas (v (catch seize grab) pres subj 2p sg)
cojamos (v (catch seize grab) pres subj 1p pl)
cojan (v (catch seize grab) pres subj 3p pl)
conozcas (v (know) pres subj 2p sg)
conozcamos (v (know) pres subj 1p pl)
conozcan (v (know) pres subj 3p pl)
parezcas (v (seem) pres subj 2p sg)
```

```
parezcamos (v (seem) pres subj 1p pl)
parezcan (v (seem) pres subj 3p pl)
venzas (v (conquer defeat) pres subj 2p sg)
venzamos (v (conquer defeat) pres subj 1p pl)
venzan (v (conquer defeat) pres subj 3p pl)
cuezas (v (cook bake) pres subj 2p sg)
cuezamos (v (cook bake) pres subj 1p pl)
cuezan (v (cook bake) pres subj 3p pl)
ejerzas (v (exercise practice) pres subj 2p sg)
ejerzamos (v (exercise practice) pres subj 1p pl)
ejerzan (v (exercise practice) pres subj 3p pl)
cruces (v (cross) pres subj 2p sg)
crucemos (v (cross) pres subj 1p pl)
crucen (v (cross) pres subj 3p pl)
```

(This is the conclusion of the laboratory.)