Massachusetts Institute of Technology
6.863J/9.611J, Natural Language Processing, Spring 2003
Department of Electrical Engineering and Computer Science
Department of Brain and Cognitive Sciences
Laboratory 1b: Two-level morphology — Building a System

Handed out: February 12, 2003 **Due: February 24, 2003**

1 Introduction

For the second part of Lab 1, you will design Kimmo automata and Kimmo lexicons that can carry out a morphological analysis of 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;

- 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? If appropriate, mention problems that arose in the design of your system.
- Pointers to your listings of your `.lex` file and your `.rul` files.
- A pointer to a log record of a recognition run on the file `spanish.rec`. This file may be found on the web page link for this laboratory.

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.)

Important: These labs have been used before and debugged. However, if you find an error in the laboratory assignment, please let us know 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. In particular, `C` stands for a possible c softening, `J` stands for possible g softening, and `Z` stands for possible z insertion. Given this, your automaton (.rul) file should be formatted as follows. Do note, however, that section 1.1.1 on page 4 discusses an alternative method of writing the automata down that most people have found preferable.

1. The very first line of the `.rul` file should declare your `ALPHABET` characters, the individual tokens that you will use.

   ```
   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 +
   ```

   Note that the affix marker '+' is required as part of the alphabet. If you later decide you need additional characters for handling Spanish phenomena, you must modify this alphabet.

2. In the next three lines, declare your `NULL` (empty, zero), `ANY` (automata relative wildcard), and `BOUNDARY` (end of word) characters:

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

3. Next declare the `SUBSETS` of characters you 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 by capital letters. 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 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

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 +
NULL 0
ANY @
BOUNDARY #
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 blank line and then are ready to specify the rule transducers themselves. Your very first automaton must define the *default* surface character correspondence (that is, the surface characters paired with themselves). It is a "dummy" automaton that has just 1 state, and one transition for each surface character, to itself.

   Thus, if there are 33 characters that can be written "on the surface," the resulting fsa table will have 33 columns. (Note that purely underlying characters like C and Z are *not* included.) This is required as a mechanical device by the Kimmo system. The dummy automaton should include the ANY character, `@`, (but *not* the affix boundary marker `+`. This is because `@` is a possible surface match, but `+` can only be an underlying, lexical character). This automaton should thus be defined as follows:

```
RULE "Surface Characters" 1 33
a ^ b c d e < f g h i { j k l m n ~ o * p q r s t u > v w x y z @
a ^ b c d e < f g h i { j k l m n ~ o * p q r s t u > v w x y z @
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 1
```

Again, if you later decide you need additional characters for handling Spanish phenomena, you must modify this automaton.

5. **Important:** if you decide to use the `fst` program to more easily write individual automaton rules, then you can skip some of the dirty work in steps 1–4 above. See below, the next section for a description of `fst`. We do urge you to try `fst` rather than typing in the ugly automata by hand.

6. You follow this "dummy" automaton with a blank like and then the 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)
.
.
```

7. 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.

If you desire more insight into what a `.rul` file should look like, please examine the file for english, `english.rul` in the course locker in the `pckimmo-old` subdirectory.

1.1.1 Building automata using fst

As the PC-KIMMO manual notes, writing out the automata by hand is something only computers, not humans, should do. "Compiling" a set rules written in so-called "re-write rule" format, e.g., "i goes to e except after c" into correct automata is In general tricky to work out, because of general and more specific applications of rules. (Also as mentioned in class, the soon-to-be extinct Xerox Corporation implemented a two-level compiler called *twolc* that could do this, and licensed their system for (a lot?) of money. In fact, as far as I understand it, they "patented" the trie-based idea for compressing dictionary storage which has then found its way into all the spelling/language handhelds that you see — e.g., Franklin dictionaries, etc. So, you see, this knowledge you are acquiring is very practical indeed!) If you want to know more about the challenging of writing a real rule compiler that "intersects" the individual automata into one, you can read the notes I will post on this subject, which have additional pointers to the literature.

To make the process of writing the rule automata easier, we have supplied a very simple rule "re-write engine", called `fst`, that takes over some of the drudgery of writing the finite-state automata tables. The program takes as input rules that are more in a minimal transition table format, and then and outputs the full finite-state tables necessary for PC-KIMMO to run as well as the required preamble before the automata. NOTE: this program is far from perfect, and sometimes requires some 'post-editing' of the tables to ensure that they work correctly. So, look over the output of this program. 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.

The `fst` program resides (surprise!) under the `fst` directory in the directory `pckimmo-old`. A simple help file can be found in that directory as `fst.help`; this is also linked on the course web site. The program takes two arguments: an input file that you write, `your-rules.fst` and writes as output the PC-Kimmo `.rul` file. To use this program then, you can cd to the fst directory and then type (for example):

```
athena% fst -o <your dir path>/turkish.rul <your dir path>/turkish.fst
```

Here is an example of what an `fst` file looks like:

```
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, but harder to edit (note how the rejecting state is automatically accommodated):

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

RULE "Bogus rule for KIMMO brain lossage" 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
```

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 a bogus subset. Note that it creates for you the "bogus rule" so that KIMMO knows that `x` can always pair with `x`.

There is an **IMPORTANT** known bug with `fst`: self-referential rules cause (not unexpectedly) infinite looping! (They are also wrong to write in the first place, however.)

Having discussed the automata format, we next turn to the Lexicon — the specification of the fsa that describes the Lexical (root and ending) tree.

1.2 Kimmo Lexicon Format

The word tree finite state automaton is written in what the system calls a `.lex` file. PC-KIMMO has a slightly baroque way of specifying the fsa "word tree". It uses *two* separate sections in the `.lex` file to describe (1) the fsa state names and then (2) the transition arcs between these fsa states. The state names are given first, in a section called `Alternations`, while the transition arcs are given in the second section, called `Lexicons`. If you find this a bit confusing, it is most helpful to take a look at the file `english.lex` in the `pckimmo-old` directory.

Your final system will have at least a main lexicon file, which may include sub-lexicons for nouns, verbs, etc. that contain the actual lexical entries for their respective roots. (In fact, you actually do not need sublexicons for this particular laboratory, but you can use them to divide up nouns and verbs for clarity.)

1.2.1 Kimmo Lexicon file

All alternation classes should be listed at the very top of the `.lex` file. What the Alternation section specifies is a list of all the state names in the word tree finite-state automaton. The very first alternation thus specifies the "root" of the lexicon tree, so it always starts out the same, as shown just below.

The format of an alternation class has three parts, all written on one line separated by spaces:

1. The keyword `ALTERNATION`
2. The *name* of the Alternation class (i.e., a state in the word tree finite-state automaton).
3. One or more state names. The intended meaning of this is that given an Alternation class, an fsa state, one can make a transition from the state label one the left to any of the state names listed on the right. However, the Alternation classes do not specify *all* the possible state transitions, nor do they provide any of the actual transition "arcs" that carry the fsa from one state to another. The remaining possible transitions as well as the actual transition labels are listed in the Lexicon section that follows the Alternations.

```
ALTERNATION Begin   <your list of state names here, e.g., Singular Plural...>
```

At least one of your states should make a transition to a special `End` state, as follows:

```
ALTERNATION            Foo            End
```

To give a more concrete example, remember that the Alternation classes are simply specifying (part of) the fsa tree transitions. For example, here's the Begin alternation from the `english.lex` file:

```
ALTERNATION           Begin    N_ROOT  ADJ_PREFIX  V_PREFIX
```

This simply says that the word search will commence by first moving to the Lexicon (state) named `N_ROOT`, (i.e., Noun roots). If we assume a serial, depth-first search, then further states after this one will be explored, with the system finally returning to state named `ADJ_PREFIX` (Adjective prefixes), and then, finally, Verb Prefixes.

Additional alternations should of course appear between the `Begin` and `End` alternations. After terminating the Alternation class specification with `End`, you skip one or more blank lines, and then go on to list the possible transitions between states. Each of the transitions emanating from a single state is labeled in the file as a `LEXICON`.

Here's how you specify `LEXICONS` in PC-KIMMO.

- First, you write the keyword `LEXICON`, followed by the name of the Lexicon (that is, a state name, as given in the `ALTERNATION` section on either the left- or right-hand sides).
- You then specify "arcs" that exit from this fsa state — the transition mapping from this state to any next states, one per line.
- Each transition takes the form of a triple: (1) the transition symbol, a string of (underlying) characters; (2) the name of a possible state (note: *not* usually the state the machine is moving to, as you'd expect); and (3) finally, any string of symbols you want printed out as a transition is made between states. This can be any string you want, but is usually something informative, and is printed out to the external world. (I know — that makes it sound like a transducer, but this is just for printing purposes.)

  Note that since the `Lexicon` refers *only* to underlying (or lexical) forms, not surface characters. Recall also that this fsa never has to worry about spelling changes — that normalization is taken care of by the spelling change finite state transducers in the `.rul` file.

The only tricky part of the arc triple description is in the second item — the state name you provide in the second column of a Lexicon arc description. There are two possibilities for this state name, depending upon the Lexicon state name itself: it is either the name of an `Alternation` class or it is not. If it is an `Alternation` class name, then the possible next states are specified by the right-hand side of the Alternation description, as we have seen in our `Begin` example above. In this case then, the `Lexicon` name and the second column of the arc specifications should *match*. However, this need not be always so. One can also specify a *different* next state than one given by the `Alternation` class descriptions by providing a single state name as the second column of the arc description.

An example will clarify. Here's an example of the case where the Lexicon name refers to an Alternation class, for English Genitives (the possessive marker 's at the end of words like `guy's`).

```
ALTERNATION Begin    N_ROOT  ADJ_PREFIX  V_PREFIX
...
ALTERNATION Genitive  End
...
END
....
LEXICON GENITIVE
+'s                    Genitive                        "+GEN"
0                      Genitive                         ""
...
```

Since the Genitive `Alternation` already specifies a next state of `End` (i.e., we must be at the end of a word), we need not specify any other next states, unless our design demands it. So, in the `Lexicon` specification, you can see that we just list `Genitive` in the second column of the transition specifications, and let the `Alternations` tell us what the next state should be (`END`). In this case, there are two character sequences that admit transitions from the Genitive state, `+'s` and `0`, The third column says what the fsa outputs as these transitions are made: if the machine successfully pairs an underlying `+'s` with some surface sequence, then it outputs `GEN` (i.e., the word is indeed Genitive). If it pairs a `0`, then it outputs nothing (this is an "otherwise" condition if all else fails).

If the `Lexicon` is *not* an Alternation class, then by definition it must appear *only* on the right-hand side of the Alternation class descriptions, and so does not have any next states defined for it there. So, you must provide these in the second column of the arc descriptions. Here's an example drawn from the English lexicon file, where we have the state name `ADJ_PREFIX` on the right-hand side of the `BEGIN` Alternation and never on the left-hand side — so we must specify the next states for `ADJ_PREFIX` in the`Lexicon` description itself. These are transitions meant to handle the prefixes of adjectives (of course!) such as *unhappy*.

```
ALTERNATION Begin    N_ROOT  ADJ_PREFIX  V_PREFIX
...
END
...
LEXICON ADJ_PREFIX
 un+           Adj_Prefix1          "NEG1"
 0             Adj_Prefix2          ""
```

In this description we see that there are two arcs that exit from the state `ADJ_PREFIX`. The first has the label `un+`. So, if this sequence of characters is successfully paired up with some surface form, then the word tree fsa can make a transition to the next state `Adj_Prefix1` (this is indeed the name of an `Alternation` class, if you look at the `english.lex` file). At the same time, the system outputs the symbols NEG1, i.e., recording the fact that "un" in front of an adjective means "NEGative of", as in *unhappy*. The second arc says that a zero (or null) symbol can carry the word tree fsa to the state Adj_Prefix2, outputting nothing.

Finally (whew!) the LEXICON section must always have two special-purpose "cleanup" transition descriptions, one that gives special treatment to the very beginning of words, kicking off their analysis, and one to take care of the very end of words, delimited by the hash mark #. (So it's a tiny lie that the Alternation classes provide *all* the state names, but you already knew that one has to fib a bit to explain things generally.)

First, there's an INITIAL LEXICON that always looks like this, and so should be at the very start of your Lexicon list, after the ALTERNATIONS:

```
LEXICON INITIAL
0                        Begin           "["
```

You should now be able to figure this out: it says that the word fsa *really* kicks off in a special initial state, and then, on processing a null symbol, goes to the state Begin, and outputs a left square bracket [.

Similarly, there is a special cleanup Lexicon for the end of a word:

```
LEXICON        End
0                         #                " ]"
```

which takes PC-KIMMO to its *real* end-of-word state denoted by a hash mark, and then spits out a closing right bracket.

This last line is always followed by a blank line and then an END keyword on a separate line, and —- you are done!

Putting this all together, your .lex file should look like this:

```
ALTERNATION           Begin ....

(your alternations here)

ALTERNATION           Foo            End

LEXICON INITIAL
0                                    Begin           "["

(your other Lexicons here)
LEXICON End
0                                        #                    " ]"

END
```

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 correctly analyze (dubbed "good examples") and also reject as ill-formed ("bad examples") in the file `spanish.rec` in the course locker. There are 47 "good" forms to handle and 13 "bad" forms in this file.

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 realize that the automata act mainly as *filters*, ruling out impossible surface/lexical 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.)

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^ 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.)