Department of Computer Science
CSC 247/447
Fall 2006

Assignment 1

Due: 3:15pm, Tuesday, September 26

Problem 1

You are to write a simplified morphological analyzer MORPH which "detaches" -s and -ed (and if you are a graduate student, -ing) endings, returning a list of pairs of possible stems along with the endings. (These are the most common endings, so it is quite useful to be able to handle them.)

For example, MORPH will give the following results (tabs/newlines added here for readability):

```
(MORPH 'houses) → ((house -s)
                   (hous  -s))
(MORPH 'tries) → ((trie -s)
                  (try  -s))
(MORPH 'bussed) → ((buss -ed)
                   (bus  -ed))
(MORPH 'skied) → ((ski  -ed)
                  (skie -ed)
                  (sky  -ed))
```

Note that not all the possible stems printed out are English words. In a complete parser, one would check each possible stem in the lexicon, and if one or more of the stems is found, the ones not found are discarded (or given low "probability").

However, if none of the possible stems are in the lexicon, then none of them can be dismissed out of hand – they are *possible* English words, even if not actual ones (*hous* is perhaps not a good candidate, since words ending in -ous are generally adjectives rather than nouns; but such cases are hard to rule out).

Here is how (MORPH INPUT) might work:

1. "Explode" INPUT into a list of characters.
2. Form BACKWORD, the reverse of the exploded INPUT.
   (So the final character is now first.)
3. If the first character of BACKWORD is not d or s (graduate students: or g), print the original INPUT and return. (I.e., the "analysis" just returns the input.)

4. If the first character of BACKWORD is *d*, then:
   If the second character of BACKWORD is not *e*,
   print INPUT and return, else

   (a) Strip *d* and *e* from BACKWORD; we will henceforth refer to the resulting list of characters as (A B C . . .) (i.e., A is the first of the remaining characters, B the second, etc.).

   (b) If A $\neq$ e, print (. . .CBA -ed); i.e., print a 2-element list in which the first element is the result of *reversing* (A B C . . .) and then "imploding" it to make a string (or atom).

   (c) If A = B and B $\in$ {b, d, g, k, l, m, n, p, r, s, t, v}, print (. . .CB -ed); i.e., delete a repeated consonant.

   (d) If A $\neq$ B and AB $\neq$ kc, print (. . .CBAe -ed); i.e., add an *e* to the stem.

   (e) If A = i, print (. . .CBy -ed); i.e., change *i* to *y*.

   (f) Return.

5. (Graduate students only.) If the first character of BACKWORD is *g*, then
   If the second and third characters of BACKWORD are not *n* and *i* respectively, print INPUT and return, else . . .

6. If the first character of BACKWORD is *s*, then
   If the second character of BACKWORD is *s*, print INPUT and return, else

   (a) Strip *s* from BACKWORD, with result (A B C . . .);

   (b) Print (. . .CBA -s);

   (c) If A = e and B $\in$ {s, z}, print (. . .CB -s);

   (d) If A = e and B = i, print (. . .Cy -s);

   (e) If A = e, B = v and C = i, print (. . .Cfe -s);

   (f) If A = e, B = v and C $\in$ {a, o, l, r}, print (. . .Cf -s);

   (g) Return.

Here are some suggested inputs:

parked, echoed, passed, picked, pulled, skied, conveyed, fizzed, patted, hugged, travel(l)ed, dial(l)ed, nonplussed, yakked, revved, smiled, confided, pulsed, died, centred (pardon my Canadian), *eyed, kneed, pirouetted, cried, studied*

shops, houses, ads, keys, buses, fizzes, trellises, tries, empties, lives, wives, knives, loaves, hooves, leaves, shelves, scarves, wolves

Perhaps you'd prefer to not explicitly break apart your input into Lisp symbols and work with the raw character vector instead. In that case it may be helpful to read Chapter 10 from Practical Common Lisp (which has already been recommended to you by Benjamin on his course page).

The given outline of the algorithm need not be rigidly adhered to. In fact, steps 4-6 would better be handled by subroutines, and certain improvements are possible.
The usual principles of good program design and documentation apply:

- the documentation should describe and explain the structure of the programs;
- the programs should contain headers briefly explaining what they do, and comments in the code where this is helpful;
- the programs should be as "elegant" as possible; i.e., they should be concise, clearly structured, and make good use of the constructs of LISP (in LISP, it is a good idea to try to "think recursively," and to use MAP functions where possible, though the present problem may not call for significant use of these functions. Also, LOOP and hashtables are generally good things to be aware of);
- the programs should be efficient, especially if there is no great price to be paid in elegance; for instance, repeated searches for membership on a fixed list of atoms can be avoided by giving each of the atoms on the list a "property" that identifies them as belonging to that list; then, instead of searching the list, we can just look for the corresponding property.
- give plenty of examples of the outputs; you might pick a paragraph of text from somewhere, and process all the words, to get some idea of how often -ed and -s endings occur and to further check the quality of the outputs; this may give you ideas for improvements.

Also, conclude with a discussion of the shortcomings of the original algorithm, any improvements you made, and remaining shortcomings of your program, and how you think a better program might be designed. How could the outputs of such a program be used for a word parser, i.e., one which gives possible tree structures for words (with morphemes at the leaves)?

Lastly, be sure to check out Benjamin's page; it contains general programming guidelines for this course, as well as how to turn your code in.