POSCLASS: An Automated Morphological Analyzer

Daniel M. Albro

June 18, 1996

1 Introduction

Upon facing a new set of language data, the morphologist is posed with a difficult task. Words must be split into morphemes, and the morphemes must be glossed and their distribution accounted for. This process can often become quite tedious and involved. The purpose of this project was to automate a portion of the task, and to provide a basis for future automations. The program described herein is intended to take as input glossed, phonemic word samples and produce as output a table of individually glossed morphemes within their position classes. For example, a file of Esperanto data might look like this:

```
esperas          [HOPE,-past,-future]
esperos          [HOPE,-past,+future]
esperadas        [HOPE,-past,-future,+continuous]
                                  ⋮
```

and the resulting table might look like this:

Position	Morphemes
1	esper ([HOPE])
2	ad ([+continuous])
3	as ([-past,-future]) os([-past,+future])

In this paper, we will discuss the desired behavior of the program and the characteristics of the actual program itself; we will then examine how closely the program comes to exhibiting the desired behavior and what changes might be desirable in the future.

2 Development

The project was divided into three stages. First, the program would take as its input a list of words that have already been divided up into morphemes, and as its output it would produce a table of position classes. For our Esperanto example, the input would look like this:

```
esper-as
esper-os
esper-ad-as
```

⋮

and the corresponding output would be:

Position	Morphemes	
1	esper	
2	ad	
3	as	os

Once position classes worked, the next stage would be to take as input a file with non-divided words paired with glosses in one-to-one correspondence with the morphemes. That is, for each morpheme in each word, there would be exactly one feature in the gloss. The Esperanto example, then, might look as follows[1]:

```
esperas       [HOPE,+present]
esperos       [+future,HOPE]
esperadas     [HOPE,+present,+continuous]
                          ⋮
```

and the result would be the following:

Position	Morphemes	
1	esper ([HOPE])	
2	ad ([+continuous])	
3	as ([+present])	os([+future])

The third (and, so far, final) stage of the project was to allow as much flexibility as possible in the specification of glosses. Thus, the Esperanto example might legitimately be as given in the first paragraph.

3 The Program

In order to understand the characteristics of the POSCLASS program, we must understand how it behaves with respect to the user (*i.e.*, its usage), how it reacts to different input data, and what algorithms were used to produce the results described.

The first thing to know, which affects all the rest, is that POSCLASS was implemented in the object-oriented scripting language Python. This language was chosen because it is a high-level language well-suited to rapid prototyping, it allows object-oriented design, it does not require compilation (and thus it saves development time), it has modules that are well-suited for the sorts of string manipulation done here, and it is available on UNIX, MS-DOS, and Macintosh computers. In the future, the code may be translated into C or C++ in order to increase the program's speed.

A second defining characteristic of POSCLASS is that it deals basically with inflectional and not derivational morphology. To the extent that derivational morphology can be made to look like inflectional morphology, the program can deal with it, but morphemes are viewed as adding features to a lexical entry in the manner of inflection rather than as changing one lexical entry into another in the manner of derivation.

[1]The order of the features should not matter

3.1 Usage

At the present time, POSCLASS is not particularly user-friendly. To use it, the user must first create a file laying out the input data. The input data file must consist of one line per word, each line consisting of a word and (optionally, if the word is pre-split) a gloss for the word. The word and its gloss must be separated from each other by white space, that is, spaces or tabs. The file may not contain anything other than such lines.

3.1.1 Word Specification

The word must be either pre-split into morphemes, with the morphemes separated by dashes, or be one long string of unindividuated morphemes. All "phonology" must have been undone. That is, the morphemes must be simply concatenated together, with no metathesis interspersing morphemes (non-concatenative morphology cannot be handled by the program as it currently exists), and no allomorphs. The words may not contain dashes (except to separate morphemes), underscores (_), spaces, tabs, or carriage returns. All other characters are acceptable.

The pre-split words are used when the program is simply taking morphemes and figuring out their position classes. In the case that pre-split words are being used, the user must be careful to disambiguate morphemes that sound the same, but are in different distribution. For example, in the language Zoque, *yah* signifies both "causative" and "plural", but the first meaning precedes the root and the second follows the root. In this case, it is necessary to write something like:

```
yah(caus)-ken-u
ken-yah(pl)-u
```

to disambiguate them.

The non-pre-split words are used when the program is taking word, gloss pairs and determining what morphemes exist, how they should be glossed, and how they are distributed. In this case, homophonous morphemes will automatically be distinguished by their glosses.

3.1.2 Gloss Specification

The gloss specification format used in POSCLASS is essentially that of the Andersonian framework. It consists of a lexical entry specifier, a set of outer features, and a set of inner features. The lexical entry specifier is a word in all capital letters indicating the root semantics of the word being glossed. For example, *DOG* might be used to indicate that the word is a member of the noun paradigm for dogs (*e.g.,* `canis [DOG,+nom]`). The outer features describe semantic or grammatical features of the word or, if the word is a verb, semantic or grammatical features of the subject of the verb. The inner features describe semantic or grammatical features of the object of a verb. Features may begin with any letter other than capital "O", and must not be comprised entirely of upper-case letters, lest they be confused with lexical entry specifiers. They may not contain commas, underscores, spaces, tabs, or carriage returns.

A gloss must be contained within square brackets, and each subpart of the gloss must be separated from the others by commas. No white space (spaces or tabs) may appear anywhere within the gloss. Inner features appear within an inner set of square brackets, of which there may be only one per gloss. Thus, a gloss must begin with a left square bracket, followed by zero or more outer features and zero or one lexical entry specifier, followed by an optional inner left square bracket, which is followed by zero or more inner features, followed by an inner right square bracket, all followed by zero or more outer features and possibly a lexical entry specifier, then finally terminated by a right square bracket. The gloss must have one and only one lexical entry specifier.

3.1.3 Running the Program

Once a data input file has been specified, the user runs the program upon that input by typing (at the command line), `posclass`, optionally followed by `-c` to indicate that only determination of position classes from pre-split morphemes is desired, followed by the name of the input file. For example, if the file *esperanto-split.inp* were to contain words in Esperanto that are split into morphemes by dashes, the user would enter `posclass -c esperanto-split.inp` to analyze the data. For a hypothetical file *esperanto-glossed.inp* that contained glossed non-split words of Esperanto, the user would enter `posclass esperanto-glossed.inp`. To save the program's output to a file, the user may add ">" followed by a filename to the end of the command line.

The output of the program is a lot of text indicating to some extent what the program is doing, followed by two position class tables indicating what the program figured out. There is usually some ambiguity as to what position class a particular morpheme belongs in, and therefore the program outputs two tables, the first indicating the leftmost possible position for each morpheme, and the second indicating the rightmost possible position for each.

3.2 Examples

This section will work through three "real-world" examples to give a clearer picture of what the program does. These examples will be in the languages Zoque, English, and Kharia.

3.2.1 Zoque

The Zoque example is actually two examples: first of the position class analyzer, and second of one-morpheme-per-feature analysis. The first example takes the paradigm for "LOOK" and splits the words into morphemes:

```
ken-u
ken-pa
y-ken-hay(ben)-u
y-ken-hay(ben)-pa
y-yah(caus)-ken-u
ken-yah(pl)-u
y-ken-hay(ben)-yah(pl)-u
y-ken-hay(ben)-t.o?y-u
ken-t.o?y-u
```

```
ken-t.o?y-pa
ken-t@?-u
y-ken-u
y-ken-pa
y-ken-hay(ben)-pa
y-ken-hay(ben)-t.o?y-u
ken-yah(pl)-t@?-u
ken-ke?t-pa
ken-ke?t-u
y-ken-hay(ben)-ke?t-u
y-ken-hay(ben)-yah(pl)-t@?-ke?t-u
ken-ke?t-pa
ken-ke?t-u-tih
ken-u-tih
y-ken-hay(ben)-u-tih
y-yah(caus)-ken-at@h-u
y-yah(caus)-ken-at@h-yah(pl)-u
na-y-ken-at@h-yah(pl)-u
na-y-ken-at@h-yah(pl)-ke?t-u-tih
hay(neg)-ken-a
hay(neg)-ken-a-tih
hay(neg)-ken-ke?t-a-tih
ken-u-a?a
ken-pa-a?a
ken-yah(pl)-u-a?a
ken-ke?t-u-a?a
ken-u-?k
ken-u-?k-a?a
ken-yah(pl)-pa-?k-a?a
ken-yah(pl)-pa-m@y
ken-u-Seh
ken-pa-mah
ken-pa-hs@?N
y-ken-hay(ben)-yah(pl)-t@?-ke?t-u-Seh-tih
y-ken-u-?k
ken-yah(pl)-ke?t-u-hs@?N
ken-u-ha
hay(neg)-ken-a-mah
hay(neg)-ken-a-a?a
hay(neg)-ken-a-hs@?N-tih
hay(neg)-ken-a-tih
ken-hay(ben)-u-a?a
ken-hay(ben)-ke?t-u-a?a
ken-pa-mah-ha
y-yah(caus)-ken-at@h-yah(pl)-t@?-u-tih
```

```
y-ken-u-a?a
y-ken-u-Seh
y-ken-ke?t-pa-tih
ken-?aNheh-u
y-ken-?aNheh-pa
y-ken-?aNheh-u-a?a
y-ken-?aNheh-yah(pl)-ke?t-u-tih
ken-yah(pl)-t.o?y-u
ken-u-a?a-hs@?N
ken-ke?t-u-a?a-Seh
y-ken-u-hs@?N-mah
y-ken-ke?t-u-a?a-tih
y-yah(caus)-ken-hay(ben)-yah(pl)-ke?t-u-?k-a?a
ken-yah(pl)-ke?t-u-Seh-tih
y-ken-ke?t-pa-tih-ha
```

The user, upon entering this data as *zoque.txt*, would run POSCLASS by entering `posclass -c zoque.txt > zoque.out`. The corresponding output (in *zoque.out*) is as follows:

```
Position classes:

Table 0

1:  na   hay(neg)
2:  y
3:  yah(caus)
4:  ken
5:  ?aNheh   hay(ben)   at@h
6:  yah(pl)
7:  t.o?y   t@?
8:  ke?t
9:  a   u   pa
10:  ?k   m@y
11:  a?a
12:  hs@?N   Seh
13:  mah   tih
14:  ha

Table 1

15:  ha   m@y
14:  tih
13:  mah   Seh
12:  hs@?N
11:  a?a
10:  ?k
```

```
9:  a   u   pa
8:  t.o?y   ke?t
7:  t@?
6:  yah(pl)
5:  ?aNheh   hay(ben)   at@h
4:  ken
3:  yah(caus)   hay(neg)
2:  y
1:  na
```

Notice that it is necessary to read the second table from bottom to top, and that *yah* and *hay* had to be disambiguated.

The second Zoque example glosses the morphemes. The words correspond to those above, but this time they are not split into morphemes, and the words are glossed as described above:

kenu	[LOOK,+past]
kenpa	[LOOK,-past]
ykenhayu	[LOOK,+ben,+past,+3serg]
ykenhaypa	[LOOK,+ben,-past,+3serg]
yyahkenu	[LOOK,+caus,+past,+3serg]
kenyahu	[LOOK,+plur,+past]
ykenhayyahu	[LOOK,+past,+plur,+ben]
ykenhayt.o?yu	[LOOK,+ben,+3serg,+desid,+past]
kent.o?yu	[LOOK,+desid,+past]
kent.o?ypa	[LOOK,+desid,-past]
kent@?u	[LOOK,+intent,+past]
ykenu	[LOOK,+past,+3serg]
ykenpa	[LOOK,-past,+3serg]
ykenhaypa	[LOOK,-past,+ben,+3serg]
ykenhayt.o?yu	[LOOK,+ben,+past,+3serg,+desid]
kenyaht@?u	[LOOK,+plur,+intent,+past]
kenke?tpa	[LOOK,-past,+repet]
kenke?tu	[LOOK,+repet,+past]
ykenhayke?tu	[LOOK,+3serg,+ben,+repet,+past]
ykenhayyaht@?ke?tu	[LOOK,+3serg,+ben,+plur,+intent,+repet,+past]
kenke?tpa	[LOOK,+repet,-past]
kenke?tutih	[LOOK,+repet,+past,+just]
kenutih	[LOOK,+past,+just]
ykenhayutih	[LOOK,+3serg,+ben,+past,+just]
yyahkenat@hu	[+3serg,+caus,LOOK,[+indef],+past]
yyahkenat@hyahu	[LOOK,+3serg,+caus,[+indef],+plur,+past]
naykenat@hyahu	[LOOK,+recip,+3serg,[+indef],+plur,+past]
naykenat@hyahke?tutih	[LOOK,+recip,+3serg,[+indef],+plur,+repet,+past,+just]
haykena	[LOOK,+neg,+negtense]
haykenatih	[LOOK,+neg,+negtense,+just]

```
haykenke?tatih            [LOOK,+neg,+negtense,+repet,+just]
kenua?a                   [LOOK,+past,+perf]
kenpaa?a                  [LOOK,-past,+perf]
kenyahua?a                [LOOK,+plur,+past,+perf]
kenke?tua?a               [LOOK,+repet,+past,+perf]
kenu?k                    [LOOK,+past,+tempsubord]
kenu?ka?a                 [LOOK,+past,+tempsubord,+perf]
kenyahpa?ka?a             [LOOK,+plur,-past,+tempsubord,+perf]
kenyahpam@y               [LOOK,+plur,-past,+locsubord]
kenuSeh                   [LOOK,+past,+similsubord]
kenpamah                  [LOOK,-past,+durative]
kenpahs@?N                [LOOK,-past,+potential]
ykenhayyaht@?ke?tuSehtih  [LOOK,+3serg,+ben,+plur,+intent,+repet,+past,+similsubord,+ju
ykenu?k                   [LOOK,+3serg,+past,+tempsubord]
kenyahke?tuhs@?N          [LOOK,+plur,+repet,+past,+potential]
kenuha                    [LOOK,+past,+interrog]
haykenamah                [LOOK,+neg,+negtense,+durative]
haykenaa?a                [LOOK,+neg,+negtense,+perf]
haykenahs@?Ntih           [LOOK,+neg,+negtense,+potential,+just]
haykenatih                [LOOK,+neg,+negtense,+just]
kenhayua?a                [LOOK,+ben,+past,+perf]
kenhayke?tua?a            [LOOK,+ben,+repet,+past,+perf]
kenpamahha                [LOOK,-past,+durative,+interrog]
yyahkenat@hyaht@?utih     [LOOK,+3serg,+caus,[+indef],+plur,+intent,+past,+just]
ykenua?a                  [LOOK,+3serg,+past,+perf]
ykenuSeh                  [LOOK,+3serg,+past,+similsubord]
ykenke?tpatih             [LOOK,+3serg,+repet,-past,+just]
ken?aNhehu                [LOOK,+complet,+past]
yken?aNhehpa              [LOOK,+3serg,+complet,-past]
yken?aNhehua?a            [LOOK,+3serg,+complet,+past,+perf]
yken?aNhehyahke?tutih     [LOOK,+3serg,+complet,+plur,+repet,+past,+just]
kenyaht.o?yu              [LOOK,+plur,+desid,+past]
kenua?ahs@?N              [LOOK,+past,+perf,+potential]
kenke?tua?aSeh            [LOOK,+repet,+past,+perf,+similsubord]
ykenuhs@?Nmah             [LOOK,+3serg,+past,+potential,+durative]
ykenke?tua?atih           [LOOK,+3serg,+repet,+past,+perf,+just]
yyahkenhayyahke?tu?ka?a   [LOOK,+3serg,+caus,+ben,+plur,+repet,+past,+tempsubord,+perf]
kenyahke?tuSehtih         [LOOK,+plur,+repet,+past,+similsubord,+just]
ykenke?tpatihha           [LOOK,+3serg,+repet,-past,+just,+interrog]
```

Notice here that there is one feature per morpheme in the input, and that the order of the features in the gloss is not significant. Notice also that object features, here *+indef* only, are specified inside square brackets. For this input file, the user would type `posclass zoque-glossed.txt` and receive (along with a great deal of preceding text (deleted) indicating what the program is doing) the following output:

```
Position classes:

Table 0

1:  hay ([+neg])   na ([+recip])
2:  y ([+3serg])
3:  yah ([+caus])
4:  ken ([LOOK])
5:  hay ([+ben])   at@h ([O+indef])   ?aNheh ([+complet])
6:  yah ([+plur])
7:  t@? ([+intent])   t.o?y ([+desid])
8:  ke?t ([+repet])
9:  a ([+negtense])   pa ([-past])   u ([+past])
10:  m@y ([+locsubord])   ?k ([+tempsubord])
11:  a?a ([+perf])
12:  Seh ([+similsubord])   hs@?N ([+potential])
13:  mah ([+durative])   tih ([+just])
14:  ha ([+interrog])

Table 1

14:  m@y ([+locsubord])   ha ([+interrog])
13:  mah ([+durative])   tih ([+just])
12:  Seh ([+similsubord])   hs@?N ([+potential])
11:  a ([+negtense])   a?a ([+perf])
10:  ?k ([+tempsubord])
9:  pa ([-past])   u ([+past])
8:  t.o?y ([+desid])   ke?t ([+repet])
7:  t@? ([+intent])
6:  yah ([+plur])
5:  hay ([+ben])   at@h ([O+indef])   ?aNheh ([+complet])
4:  ken ([LOOK])
3:  hay ([+neg])   yah ([+caus])
2:  y ([+3serg])
1:  na ([+recip])
```

Each morpheme receives a single feature as its gloss, and the tables come out more or less the same as in the pre-split example, with some slight shifts due to different processing orders. Note that object features are indicated by a preceding capital "O", which is why input features may not begin with "O".

3.2.2 English

We will now show how POSCLASS can learn the present tense verb system of English. This example, while much shorter than the previous, illustrates a few loosenings of the one-gloss-per-morpheme rule. Here, we show that POSCLASS can handle identical words with different glosses and multiple features per morpheme. It also shows that the program can handle multiple paradigms (in this case, “LOOK” and “COOK”) in one file. The input file, *english.txt*, is as follows:

```
look            [LOOK,+me,-you,-plur]
look            [LOOK,-me,+you,-plur]
looks           [LOOK,-me,-you,-plur]
look            [LOOK,+me,-you,+plur]
look            [LOOK,-me,+you,+plur]
look            [LOOK,-me,-you,+plur]
cook            [COOK,+me,-you,-plur]
cook            [COOK,-me,+you,-plur]
cooks           [COOK,-me,-you,-plur]
cook            [COOK,+me,-you,+plur]
cook            [COOK,-me,+you,+plur]
cook            [COOK,-me,-you,+plur]
```

The corresponding output table is as follows:

```
1:  cook ([COOK])   look ([LOOK])
2:  s ([-me,-you,-plur])

Table 1

2:  s ([-me,-you,-plur])
1:  cook ([COOK])   look ([LOOK])
```

3.2.3 Kharia

The Kharia example is perhaps the most complicated of all. In it, there are overlapping feature specifications (some morphemes are specified by sets of features whose intersection is not empty). The input data consists of a partial paradigm of *gil* “to beat”:

```
giliN           [BEAT,-past,-perf,-habit,-futvp,-him,-you]
gilem           [BEAT,-past,-perf,-habit,-futvp,-him,+you]
gile            [BEAT,-past,-perf,-habit,-futvp,+him,-you]
giltiN          [BEAT,-past,-perf,+habit,-futvp,-him,-you]
giltem          [BEAT,-past,-perf,+habit,-futvp,-him,+you]
gilte           [BEAT,-past,-perf,+habit,-futvp,+him,-you]
giloj           [BEAT,+past,-perf,-habit,-futvp,-him,-you]
gilob           [BEAT,+past,-perf,-habit,-futvp,-him,+you]
gilog           [BEAT,+past,-perf,-habit,-futvp,+him,-you]
gilsigiN        [BEAT,-past,+perf,-habit,+futvp,-him,-you]
gilsigem        [BEAT,-past,+perf,-habit,+futvp,-him,+you]
gilsige         [BEAT,-past,+perf,-habit,+futvp,+him,-you]
gilsigDiN       [BEAT,-past,+perf,-habit,-futvp,-him,-you]
gilsigDem       [BEAT,-past,+perf,-habit,-futvp,-him,+you]
```

```
gilsig        [BEAT,-past,+perf,-habit,-futvp,+him,-you]
gilsigtiN     [BEAT,-past,+perf,+habit,-futvp,-him,-you]
gilsigtem     [BEAT,-past,+perf,+habit,-futvp,-him,+you]
gilsigte      [BEAT,-past,+perf,+habit,-futvp,+him,-you]
gilsig'oj     [BEAT,+past,+perf,-habit,-futvp,-him,-you]
gilsig'ob     [BEAT,+past,+perf,-habit,-futvp,-him,+you]
gilsig'og     [BEAT,+past,+perf,-habit,-futvp,+him,-you]
```

The output table is as follows:

```
Position classes:

Table 0

1:  gil ([BEAT])
2:  sig ([+perf])
3:  ' ([+past,+perf])   t ([+habit])   D ([-past,+perf,-habit,-futvp,-him])
    e ([+futvp,+him])
4:  o ([+past])
5:  iN ([-past,-him,-you])   em ([-past,+you])   b ([+past,+you])
    e ([+perf,+habit,+him])   e ([-past,-perf,+him])   g ([+past,+him])
    j ([+past,-him,-you])

Table 1

5:  iN ([-past,-him,-you])   em ([-past,+you])   b ([+past,+you])
    e ([+perf,+habit,+him])   e ([-past,-perf,+him])   g ([+past,+him])
    j ([+past,-him,-you])   e ([+futvp,+him])
4:  D ([-past,+perf,-habit,-futvp,-him])   o ([+past])
3:  ' ([+past,+perf])   t ([+habit])
2:  sig ([+perf])
1:  gil ([BEAT])
```

Note that the morpheme *e* shows up three times in the output, even though it is in some sense the same morpheme. This is because the actual distribution is something like "*e* appears as the exponent of *-past,+him* in all cases except the present perfect", but the program does not handle exceptions, so it just lists all of the different places where *e can* appear. The other thing to note is that the program does not include redundant features. One might want to say, for example, that *em* signifies *[-past,-him,+you]* rather than simply *[-past,+you]*, but the latter is sufficient to characterize the distribution of *em*, so the program conservatively chooses the latter. This points out the fact that users of the program should not take its output as the gospel truth, but rather look to see if slight variations might be appropriate. For example, with the Zoque data, the user might want to use the glosses to combine the two position class tables into a single table that puts morphemes with similar meanings into the same classes wherever possible, and here, the user might want to add features to the glosses for each morpheme.

3.3 Internals

We will now move from what the program does to how it does it. The actual code can be seen in Appendix A. First, we will look at how morphemes are arranged into position classes, and then we will move on to analysis of word, gloss pairs.

3.3.1 Position Class Analysis

Position class analysis takes as its input pre-split words and outputs a position class table. It uses an incremental algorithm with an order of growth roughly linear with the number of input words. That is, it fully processes each word as it comes in and does not need to remember previously-heard words. For each word, the program produces a list of morphemes by dividing around the dashes and then updates two position class tables with the list of morphemes. The order of the morphemes within a word is presumed to be mandatory; that is, if the morphemes in a word appear in a given order in the input, it is assumed that no other order of those morphemes is grammatical.

Internally, the two position class tables are stored as a single lookup-list, where each morpheme is matched with a pair of values: the position class in the first table, and the position class in the second table. The first table contains the leftmost possible position for each morpheme, and the second table contains the rightmost. The way this is done is to update the first table with the list of morphemes generated by splitting up a word, then reverse the order of morphemes and use the same code to update the second table. Thus, the table update code always puts each morpheme as far to the left as it can, and the second table is produced by looking at each word backwards.

The table update code works as follows. Loop over the morphemes in the word. If the current morpheme has not yet been entered into the table, record that it has no bounding elements. Look at the morphemes that are coming up after this morpheme in the current word. If one of them has already been entered, put the current morpheme in the table to the left of all already entered morphemes from the current word that are to the right of the current morpheme in the current word, but just to the right of the previous morpheme entered from the current word, if any. If necessary, bump the higher morphemes up to leave room for this one and note that the current morpheme is a left bound for them. If none of the upcoming morphemes has yet been entered, however, enter the current morpheme just after the previous morpheme entered from the current word, if any, and note that the previously entered morpheme is a left bound for the current morpheme. If, however, the current morpheme had already been entered into the table, check to make sure that its position in the current word is consistent with its position in the table. If it is not consistent, output an error message and ignore the current morpheme. Otherwise, if the previously recorded position of the current morpheme is less than or equal to the position of the previously recorded morpheme, move the current morpheme just after the previously recorded one and record the previously recorded morpheme as a left bound for the current one. If necessary, bump up the morphemes to the right of the previously entered morpheme to make room for the current one (this is necessary if one of the morphemes to the right is a bound for the current morpheme in the other table or if the current morpheme is a left-boundary for one of the morphemes to the right). Move on to the next morpheme and do the same, until all of the morphemes in the word have been processed.

Written as an algorithm, the above looks as follows:

```
for morph in morphemes:
   if morph not yet entered:
      then note: morph has no bounding elements
           guess: no upcoming morphemes are in the table
                  after the last entered
           for upcoming in morphemes after morph:
              if upcoming has been entered already:
                 then if upcoming is in the table just after the last entered
                    then bump all morphemes above upcoming up one
                         bump upcoming up one
                         place morph in the position class where upcoming was.
                         note: morph is a left bound for upcoming
                         note: there was a morpheme like we guessed there wasn't.
                         break out of the loop.
                 end if
              end if
           next upcoming
           if morph hasn't been inserted yet:
              then note: the previous morpheme is a bound for morph
                   place morph just after the previously inserted morpheme
           end if
      else if in the first table and the placement of morph is inconsistent:
           then output error message
                continue on to the next morpheme
          end if
          if the previous position of morph is at or before
             the position at which a morpheme was last added
             then note: the previously entered morpheme is a bound for morph
                  if morph entering the position after the previously entered morpheme
                     would violate recorded boundaries
                     then bump everything at that position and above right
                  end if
                  place morph at the position after the previously entered morpheme
          end if
   end if
next morph
```

3.3.2 Word, Gloss Pair Analysis

In order to analyze word, gloss pairs into morphemes with position classes, POSCLASS reads all of the word, gloss pairs in the input file into a single list. It then splits up the gloss representations into lists of features, with the inner features marked with an initial "O" and reünites the gloss lists back with the corresponding words. Thus, for example, `yyahkenatehu [+3serg,+caus,LOOK,[+indef],+past]` turns into `('yyahkenatehu', ['+3serg', '+caus', 'LOOK', 'O+indef', '+past'])`.

The program then loops through each word, gloss pair in turn. It figures out which morpheme corresponds to the root of the word by finding the greatest common substring of all words glossed as having the same lexical entry specification as the current word. The program then adds the root to the position class chart by sending the position class analyzer (described above) a word containing just the root. Note that in the word, gloss pair analysis part of the program, the words sent to the position class analyzer will always consist of morphemes followed by parenthetical glosses. For example, the root for "LOOK" in Zoque would be sent to the position class analyzer as "ken ([LOOK])".

Once the root has been found, the program tries out every conceivable combination of the remaining features, making a paradigm for each combination and trying to see whether a single morpheme corresponds to any of the feature combinations. Feature combinations are tried in the order left-appearing features before right-appearing, smaller combinations before larger. Feature combinations that have been tried before are ignored. Essentially, the program collects all of the words that have a particular combination of features and then tries to see what the greatest common substring of the word list is, with the greatest common substring search being limited to "unanalyzed material"—that is, the parts of each word that have not already been identified as associated morphemes. For example, if *yah*, *ken*, and *y* have already been identified, and we are looking for the morpheme corresponding to *+continuous*, we take all of the words marked as *+continuous*, subtract out *yah*, *ken*, and *y* from each of them, and find the largest common substring from the remainder. If one and only one greatest common substring is found corresponding to a given feature combination, it is chosen as the morpheme corresponding to that feature combination. Each word in the paradigm is then glued back together in the original order, using only the previously analyzed morphemes and the new morpheme, and sent to the position class analyzer to update the position classes.

The algorithm for analyzing word, gloss pairs has an order of growth in time that rises exponentially (factorially) with the average number of features assigned to each word, due to the necessity to check each possible feature combination for each word.

4 Results and Discussion

Now that we have seen how the program as it currently exists works, we can examine what it's limitations are. Currently, the program is perhaps overly conservative. For example, if a particular morpheme showed up whenever the features *-me, -you* showed up, and the features *-me, -you* only appeared together, the program would guess that *-me* was responsible for the morpheme rather than both of them. It also does not handle cases where the presence of a particular feature combination causes a morpheme not to appear, for example in Georgian, where only one morpheme is allowed as a prefix or suffix to each stem, so there is an ordered hierarchy of features—if the most privileged set of features is present, then the morphemes corresponding to less privileged features don't appear. Finally, the program does not handle cases where one feature corresponds to multiple morphemes, as in a circumfix situation.

In the future, several modifications to the program might be possible and desirable. First of all, of course, it would be nice to fix the shortcomings listed above. However, it could very well be the case that fixing them, if even possible, would involve a total rewrite of the algorithms involved. In addition to fixing the shortcomings above, however, there are many capabilities that could be added to the program. For one thing, a nice user interface could be added, and for another, extended capabilities could be added. For example, if POSCLASS were to be combined with a program such as KIMMO, one could take actual phonetic transcriptions of words, use KIMMO to reverse the phonology, and then use POSCLASS to analyze the morphology. An even more ambitious plan might be to automatically analyze the phonology. For example, if the phonetic form were read into an autosegmental tree structure via the algorithm used in the AMAR program, greatest common substrings could be computed by a "sloppy" algorithm that mandates simply that the substrings have most of the same features and connections. This would eliminate most of the problems of allomorphy, while not requiring an explicit abstract underlying representation. Another possible modification might be to extend the program to automatically output Andersonian disjunctive blocks and rules instead of position classes.

A Code

A.1 posclass

```
#!/usr/local/bin/python

from posclass import *

def output_message():
    print 'Usage:'
    print '\tposclass [-c|-p] <file>'
    print 'where:'
    print '\t-c\tsignifies that we are to find position classes from pre-split morphemes,'
    print '\t-p\tsignifies that we are to parse unsplit morphemes, and'
    print '\t<file>\tis the input data file.'
    print

prs = MorphemeParser()

if len(sys.argv) == 2:
    if sys.argv[1][0] == '-':
        output_message()
    else:
        prs.parse_morphemes(sys.argv[1])
elif len(sys.argv) == 3:
    if sys.argv[1] == '-c':
        prs.get_posclasses(sys.argv[2])
    elif sys.argv[1] == '-p':
        prs.parse_morphemes(sys.argv[2])
    else:
        output_message()
else:
    output_message()
```

A.2 posclass.py

```
# $Id: posclass.py,v 1.7 1996/06/17 23:57:28 albro Exp albro $
# --------
import sys
import string
import regex
import tparsing
```

```
# Regular expression that separates out a morpheme and its glosses
# from a string containing both.  For example, a matching
# string might be "bob ([+human,+noun,+name])".
#      morph_rx.match("bob ([+human,+noun,+name])") would
#      return something other than -1, and then
#      morph_rx.group(1) would return "bob",
#      morph_rx.group(2) would return "[+human,+noun,+name]".
MORPH_REGX = '^\([^ ]+\) (\[\(.+\)\])$'
morph_rx = regex.compile(MORPH_REGX)

# Regular expression that matches a string entirely composed of
# one or more uppercase letters.
UPC_REGX = '^['+string.uppercase+']+$'
upc = regex.compile(UPC_REGX)

# Regular expression to find non-whitespace strings within
# square brackets.  (Entire string must be of the form
# [......])
STRSQ_REGX = '^\[\([^'+string.whitespace+']+\)\]$'
sqbraks = regex.compile(STRSQ_REGX)

# Matches strings with a left square bracket in them.
SIMPL_REGX = '.*\[.*'
simpl = regex.compile(SIMPL_REGX)

# In a line of the form ...[...]..., figures out what
# precedes the square brackets, what is contained within them,
# and what follows them.
INRSQ_REGX = '^\(.*\)\[\(.*\)\],?\(.*\)$'
inrsqbraks = regex.compile(INRSQ_REGX)

class PosClassTable:
    """Class that handles position classes.

    Handles position classes as two tables, one wherein the
    morphemes are put as far left in the table as they could
    possibly go, and one wherein the morphemes are put as far
    right in the table as they could possibly go.  With enough
    data, this representation settles out into a single consistent
    table.
    """
    posclass = {} # The actual data, morpheme -> (tbl1, tbl2)
    bounds = {}   # The bounding morphemes for each morpheme. ([], [])
                  # As a tuple left-bounds, right-bounds.

    def insert_at(self, pos, the_morph, tbl):
        """
        Insert the_morph at pos in tbl, moving all others above that
        point upwards.
        """
        for morph in self.posclass.keys():
            position = self.posclass[morph]
            if position[tbl] > pos:
                self.set_class(morph, position, position[tbl]+1, tbl)
        if self.posclass.has_key(the_morph):
            position = self.posclass[the_morph]
            self.set_class(the_morph, position, pos, tbl)
        else:
            position = (-1, -1)
            self.set_class(the_morph, position, pos, tbl)
        self.last_used = pos

    def bump_right(self, pos, tbl):
        """ Move everything in tbl at or above pos to the right.
        """
        for morph in self.posclass.keys():
            position = self.posclass[morph]
            if position[tbl] >= pos:
                self.set_class(morph, position, position[tbl]+1, tbl)

    def set_class(self, morph, position, new_pos, tbl):
        """Set the position class of morph in tbl to be new_pos.

        position should be the previous position class (as a tuple
        of table0, table1), or (-1, -1) if there was no previous
        position class.
        """
        if tbl == 0:
            self.posclass[morph] = (new_pos, position[1])
        else:
            self.posclass[morph] = (position[0], new_pos)

    def has_real_key(self, morph, tbl):
        """Returns 1 if morph has a true, specified position class in
        table tbl, otherwise returns 0
        """
        if self.posclass.has_key(morph):
            if self.posclass[morph][tbl] > -1:
                return 1
        return 0

    def can_infringe(self, morph1, morph2, tbl):
        """Can morph1 move into a box with morph2 in tbl?
        """
        # Cannot if morph2 is a bound for morph1 in the other
        # table.  Also cannot if morph1 is a bound for morph2
        # in the *same* table.
        bounds1 = self.bounds[morph1]
        bounds2 = self.bounds[morph2]

        other = 1
        if tbl == 1:
            other = 0

        if morph2 in bounds1[other]:
            return 0
        else:
            if morph1 in bounds2[tbl]:
                return 0
            else:
```

```
            return 1

    def can_enter_box(self, morph, box, tbl):
        """Can morph move into box box in tbl?
        """
        # We can do it only if morph can infringe upon all of the
        # elements of box.
        can_enter = 1
        for occ in self.posclass.keys():
            if self.posclass[occ][tbl] == box:
                if not self.can_infringe(morph, occ, tbl):
                    can_enter = 0
                    break
        return can_enter

    def morpheme_in_disguise(self, idx, morphs):
        """Detects whether the morpheme at idx is in conflicting
        distribution with earlier records.  This indicates that
        one of the morphemes is actually two morphemes that are
        pronounced identically.  Returns a list of morphemes
        that might be in disguise.
        """
        mi = morphs[idx]

        if not self.bounds.has_key(mi):
            return []

        # Appears right of an element of the right bounds?
        left = morphs[0:idx]
        for morph in left:
            if morph in self.bounds[mi][1]:
                return [morph]

        # Appears left of an element of the left bounds?
        right = morphs[idx+1:len(morphs)]
        for morph in right:
            if morph in self.bounds[mi][0]:
                return [morph]

        # No?  Then it's okay.
        return []

    def update_table(self, tbl, morphs):
        """Updates the position class table number tbl with
        the data from morphs, which is an ordered list of the
        morphemes in an input word.
        """
        self.last_used = 0
        i = 0
        while i < len(morphs):
            if not self.has_real_key(morphs[i], tbl):
                self.bounds[morphs[i]] = ([], [])
                inserted = 0
                for j in range(i+1, len(morphs)):
                    if self.has_real_key(morphs[j], tbl):
```

```
            # Make sure it’s after the last one, and
            # before this one, but as far left as possible
            pos = self.posclass[morphs[j]]
            if pos[tbl]-1 == self.last_used:
                self.insert_at(pos[tbl], morphs[i], tbl)
                self.set_class(morphs[j], pos, pos[tbl]+1, tbl)
                self.bounds[morphs[j]][tbl].append(morphs[i])
                inserted = 1
            break
    if inserted == 0:
        if self.posclass.has_key(morphs[i]):
            pos = self.posclass[morphs[i]]
        else:
            pos = (-1, -1)
        if self.last_used > 0:
            self.bounds[morphs[i]][tbl].append(morphs[i-1])
        self.last_used = self.last_used + 1
        self.set_class(morphs[i], pos, self.last_used, tbl)
else:
    # Here, we should check to see if it is actually another
    # morpheme in disguise --- it is if it appears explicitly
    # left of an element of its left bounds or explicitly right
    # of its right bounds.
    # The morphemes that preceded morphs[i] = morphs[0:i]
    # The morphemes that follow morphs[i] = morphs[i+1:len(morphs)]
    if tbl == 0:
        mids = self.morpheme_in_disguise(i, morphs)
        if len(mids) > 0:
            print "The relative positions of morphemes",
            print "’%s’ and ’%s’ are" % (morphs[i], mids[0]),
            print "ambiguous.  This"
            print ’signifies that there are actually two’,
            print ’morphemes with the same phonological form,’
            print "being either ’%s’ or ’%s’. " % (morphs[i],
                                                   mids[0]),
            print "These should be distinguishable by the",
            print "semantics,"
            print ’so please disambiguate the offending morpheme’,
            print ’in’
            print ’the input and rerun.  Meanwhile, the morpheme’,
            print ’will be ignored.’
            i = i + 1
            continue

    # Back to the normal code.
    pos = self.posclass[morphs[i]]
    if pos[tbl] <= self.last_used:
        self.bounds[morphs[i]][tbl].append(morphs[i-1])
        self.last_used = self.last_used + 1
        if not self.can_enter_box(morphs[i], self.last_used, tbl):
            self.bump_right(self.last_used, tbl)
        self.set_class(morphs[i], pos, self.last_used, tbl)
    else:
        self.last_used = pos[tbl]
```

```
            i = i + 1

    def process(self, word):
        """
        Collect precedence data from word, which is a word divided
        into morphemes by the separator '_'
        """
        print 'Processing word', word

        morphs = string.split(word, '_')

        self.update_table(0, morphs)
        morphs.reverse()
        self.update_table(1, morphs)

    def table_max(self, tbl):
        """Returns the maximum position class in tbl.
        """
        max = 0
        for morph in self.posclass.keys():
            if self.posclass[morph][tbl] > max:
                max = self.posclass[morph][tbl]
        return max

    def display(self):
        """Outputs a human-readable representation of the position
        class tables.
        """

        print 'Position classes:'
        print
        for tbl in range(2):
            print 'Table', tbl
            print

            max = self.table_max(tbl)
            for i in range(1, max+1):
                if tbl == 0:
                    print '%d: ' % i,
                else:
                    print '%d: ' % (max - i + 1),
                for morph in self.posclass.keys():
                    if self.posclass[morph][tbl] == i:
                        print morph, ' ',
                print

            if tbl == 0:
                print
            else:
                print '-----------'

    def split_word(self, word, glosses):
        """
        Split up *word*, finding the parts that are in or not in
        the position class table.
        """
        # Collect a list of the pre-discovered morphemes already there.
        in_it_list = []
        for key in self.posclass.keys():
            result = morph_rx.match(key)
            if result == -1:
                print 'Error: Unidentified morpheme', key
                sys.exit(1)
            else:
                morph = morph_rx.group(1)
                gloss = morph_rx.group(2)

            key_glosses = string.split(gloss, ',')
            if string.find(word, morph) >= 0:
                # To be excluded, it either has to have the right
                # glosses or be the root morpheme.
                add_it = 1

                # First, test if it is the root morpheme:
                if upc.match(gloss) == -1:
                    # Not the root morpheme.  Therefore test
                    # to see if el is in the glosses.
                    for el in key_glosses:
                        if el not in glosses:
                            add_it = 0
                            break

                if add_it == 1:
                    in_it_list.append((morph, key_glosses, key,
                                       self.posclass[key][0]))

        # Make a template based on the position classes.
        key_list = []
        tmplt = '_'
        max = self.table_max(0)
        for i in range(1, max+1):
            for el in in_it_list:
                if el[3] == i:
                    tmplt = tmplt + el[0] + '_'
                    key_list.append(el[2])

        # Use it!
        T = tparsing.Template(tmplt, '_')
        try:
            (matches, lastidx) = T.PARSE(word)
        except ValueError, msg:
            print 'Something weird:', msg
            sys.exit(1)

        return (matches, key_list, tmplt)

class Glosses:
    """
    Class that keeps track of a paradigm (all of the forms related to
    a single 'semantic' entity - e.g., DOG).  Contains methods that
```

```
use the paradigm information to figure out information about the
morphology.
"""
glosses = []
analyzed= {}
morphs = {}
submitted = {}

def __init__(self, pclass):
    """Method called upon creation of a class object.  lex is the
    lexical entry name, e.g. 'LOOK'
    """
    self.glosses = []
    self.analyzed = {}
    self.morphs = {}
    self.submitted = {}
    self.pclass = pclass

def add_gloss(self, word, gloss):
    """Adds a word-gloss pair to the paradigm.
    """
    self.glosses.append((word, gloss))

def display(self):
    """Outputs a vaguely human-readable form of the paradigm.
    """
    print '--------'
    print 'Glosses:'
    for el in self.glosses:
        print '   %s: ' % el[0],
        print el[1]

def greatest_common_strings(self, list):
    """
    Find the greatest common strings of list.  That is, return
    the set of strings that are found as substrings of all strings
    in list, where the set only contains the largest possible such.
    """

    # Determine the shortest string in the list
    min = sys.maxint
    mel = ''
    for el in list:
        if len(el) < min:
            min = len(el)
            mel = el

    # Test each permutation, descending downward.
    results = []
    width = min
    success = 0
    while width > 0:
        i = 0
        j = i + width
        while j <= min and success == 0:
                in_all = 1
                for el in list:
                    if string.find(el, mel[i:j]) < 0:
                        in_all = 0
                        break
                if in_all != 0:
                    results.append(mel[i:j])
                    success = 1
                i = i + 1
                j = i + width
            width = width - 1
        if success == 0:
            for i in range(len(mel)):
                in_all = 1
                if string.find(el, mel[i]) < 0:
                    in_all = 0
                    break
                if in_all != 0:
                    results.append(mel[i])
                    success = 1

        return results

    def gcs_multiples(self, mlist):
        """
        Find the greatest common strings of list.  That is, return
        the set of strings that are found as substrings of all strings
        in list, where the set only contains the largest possible such.
        """

        # Make new list:
        list = []
        for el in mlist:
            list.append(string.join(el[0], '_'))

        # Determine the shortest string in the list
        min = sys.maxint
        mel = ''
        for el in list:
            if len(el) < min:
                min = len(el)
                mel = el

        # Test each permutation, descending downward.
        results = []
        width = min
        success = 0
        while width > 0:
            i = 0
            j = i + width
            while j <= min and success == 0:
                if string.find(mel[i:j], '_') < 0:
                    in_all = 1
                    for el in list:
                        if string.find(el, mel[i:j]) < 0:
```

```
                        in_all = 0
                        break
                if in_all != 0:
                    results.append(mel[i:j])
                    success = 1
            i = i + 1
            j = i + width
        width = width - 1

    if success == 0:
        for i in range(len(mel)):
            if mel[i] != '_':
                in_all = 1
                if string.find(el, mel[i]) < 0:
                    in_all = 0
                    break
                if in_all != 0:
                    results.append(mel[i])
                    success = 1

    return results

def min_tuple(self, size, tuple):
    """Return the minimal indices corresponding to tuple within size.
    """
    indices = []
    for i in range(size):
        indices.append(-1)

    for i in range(tuple):
        indices[i] = i
    indices[tuple] = tuple - 1

    return indices

def next_indices(self, indices, size, tuple):
    """Returns the next set of indices of a permutation.

    Example: if size is 3, should return [0],[1],[2],[0,1],
             [0,2], [1,2], [0,1,2], in that order.
    (Actually, [0,-1,-1], [1,-1,-1], [2,-1,-1],...)
    """
    while tuple < size:
        for i in range(tuple, -1, -1):
            indices[i] = indices[i] + 1
            if indices[i] < size:
                for j in range(i+1, tuple+1):
                    indices[j] = indices[j-1] + 1
                if indices[tuple] < size:
                    return (indices, tuple)
        if indices[0] >= (size - 1):
            tuple = tuple + 1
            if tuple < size:
                indices = self.min_tuple(size, tuple)
    # If we get this far, this means there aren't any more
    # possibilities
    return ([], -1)

def display_analyzed(self):
    print 'Analyzed Features =',

    i = 1
    for key in self.analyzed.keys():
        if i == 4:
            print
            i = 1

        if len(self.analyzed[key][0]) > 0:
            print key, '=', self.analyzed[key][0], ';',
            i = i + 1

    print '...'

def parse_word(self, word, the_glosses):
    """Given a fully built-up paradigm, figure out what morphemes
    are in word and where the morpheme boundaries are.  Use this
    information to update the position classes.
    """

    glist = the_glosses[:]

    # Remove the paradigm identifier
    for i in range(len(glist)):
        if upc.match(glist[i]) != -1:
            self.try_features(glist[i], [glist[i]])
            del glist[i]
            break

    print 'Glosses =', glist
    self.display_analyzed()
    print 'Morphs =', self.morphs

    if len(glist) < 1:
        return

    # Now try to analyze everything else.
    tuple = 0
    size = len(glist)
    indices = self.min_tuple(size, tuple)

    (indices, tuple) = self.next_indices(indices, size, tuple)
    while tuple > -1:
        test = []
        for i in range(tuple+1):
            test.append(glist[indices[i]])

        features = string.join(test, ',')
        self.try_features(features, test)
        (indices, tuple) = self.next_indices(indices, size, tuple)

def try_features(self, features, test):
    """Test a particular set of features against a word to extract
    morphemes.

    *features* is a string representation, *test* is the list of
    features.
    """
    if not self.analyzed.has_key(features):
        min_word, paradigm = self.test_features(test)
        if len(min_word) == 0:
            self.analyzed[features] = ('', paradigm)
        else:
            print 'Found Morpheme', min_word
            result = morph_rx.match(min_word)
            if result == -1:
                print 'Internal Error'
                sys.exit(1)

            simple = morph_rx.group(1)
            self.analyzed[features] = (simple, paradigm)

            if self.morphs.has_key(simple):
                ftr_list = self.morphs[simple]
                found_one = 0
                idx = 0
                print 'This paradigm is', paradigm
                for ftr in ftr_list:
                    print 'Paradigm for', simple, ftr, 'is',
                    print self.analyzed[ftr][1]

                    if self.analyzed[ftr][1] == paradigm:
                        # Same paradigms, therefore this is
                        # a new feature...
                        # Do something about that.
                        found_one = 1
                        break
                    idx = idx + 1

                if found_one:
                    print 'Added a spec.'
                    if self.pclass.posclass.has_key(min_word):
                        pc = self.pclass.posclass[min_word]
                        old_word = simple + ' ([' + ftr + '])'
                        del self.pclass.posclass[old_word]
                        del self.pclass.posclass[min_word]
                        newwrd = simple + ' ([' + ftr + features + '])'
                        self.pclass.posclass[newwrd] = pc
                        self.analyzed[ftr+features] = (newwrd,
                                                       paradigm)
                        self.morphs[simple][idx] = ftr+features
                else:
                    self.morphs[simple].append(features)
            else:
                self.morphs[simple] = [features]

    def assemble_paradigm(self, test):
        """Assemble a paradigm including the features in test.
        """
        paradigm = []
        for gpair in self.glosses:
            all_in = 1
            for el in test:
                if el not in gpair[1]:
                    all_in = 0
                    break
            if all_in == 1:
                paradigm.append(gpair)

        return paradigm

    def test_features(self, test):
        """Test the features in test to see if they correspond to a
        unique morpheme in the paradigm.  If so, find where the morpheme
        fits into the words in which it is found, and submit the
        words to the position class analyzer.  Note that only identified
        morphemes will be sent.
        """
        # 1. Assemble a paradigm including the features in test.
        paradigm = self.assemble_paradigm(test)

        # 2. X-out the parts (morphemes) of the paradigm that have already been
        #    found/analyzed.
        str_list = []
        for el in paradigm:
            str_list.append(self.pclass.split_word(el[0], el[1]))

        # 3. Find the greatest common strings in the paradigm.
        gcs = self.gcs_multiples(str_list)

        # 4. Resubmit the paradigm members with the new morpheme to the
        #    MorphemeParser object to help determine position classes.
        if len(gcs) == 1:
            # There was one and only one corresponding morpheme.
            # Therefore, we will submit it as the morpheme
            # corresponding to the features.
            newmorph = gcs[0] + ' (['+string.join(test, ',')+'])'

            # Check to see if any full matches are available.
            full_matches_available = 0
            for el in str_list:
                for i in range(len(el[0])):
                    if el[0][i] == gcs[0]:
                        full_matches_available = 1
                        break
                if full_matches_available == 1:
                    break

            for el in str_list:
                if string.find(el[2], '_'+gcs[0]+'_') == -1:
                    pos = -1
                    # Try first to find an exact match:
                    for i in range(len(el[0])):
                        if el[0][i] == gcs[0]:
                            pos = i
                            break
                    if pos == -1 and full_matches_available == 0:
                        # Then settle for a partial one.
                        for i in range(len(el[0])):
                            if string.find(el[0][i], gcs[0]) != -1:
                                pos = i
                                break
                    if pos != -1:
                        new_word = ""
                        for i in range(pos):
                            new_word = new_word + el[1][i] + '_'
                        new_word = new_word + newmorph
                        for i in range(pos, len(el[1])):
                            new_word = new_word + '_' + el[1][i]
                        if not self.submitted.has_key(new_word):
                            self.pclass.process(new_word)
                            self.submitted[new_word] = None

            return (newmorph, paradigm)
        else:
            # The features didn't correspond to anything.  Return
            # the empty string.
            return ('', paradigm)

    def find_morphemes(self):
        """Figure out all of the morphemes in the paradigm.

        This is the top-level function for this class.
        """
        # Figure out the morphemes in each word.
        for el in self.glosses:
            print 'Parsing morphemes in', el[0]
            self.parse_word(el[0], el[1])
            self.pclass.display()

class MorphemeParser:
    """
    Class that does morpheme parsing.
    """

    def __init__(self):
        """Called upon creation of an object of this class.
        """
        self.pclass = PosClassTable()   # Create a position class object.
        self.paradigms = Glosses(self.pclass)

    def get_posclasses(self, file):
        """Given a file named *file* which contains a number of lines,
        each line containing as its leftmost element a word split into
        morphemes by dashes, determines the position classes.
        """
        # Open the file.
        try:
            f = open(file, 'r')
        except IOError:
            print 'Error opening file.\n'
            sys.exit(1)

        # Read and process each line
        try:
            ln = f.readline()
            while ln != '':
                words = string.split(ln)
                if len(words) != 0:
                    word = string.join(string.split(words[0], '-'), '_')
                    self.pclass.process(word)
                ln = f.readline()
        except EOFError:
            print 'End.'

        # Display the result.
        self.pclass.display()

        # Close the file.
        f.close()

    def parse_glosses(self, list):
        """
        Parse the glosses in list and enter them into the corresponding
        paradigm.  List contains pairs of (word, gloss).
        """

        new_list = []
        lineno = 0
        for el in list:
            # Keep track of the line number, for error display purposes.
            lineno = lineno + 1

            # Make sure that the gloss is inside square brackets.
            result = sqbraks.match(el[1])
            if result == -1:
                print 'Malformed gloss in line', lineno
                sys.exit(1)

            # Get the stuff inside the square brackets.
            inside = sqbraks.group(1)

            # Check to see if there is an object specification.
            result = simpl.match(inside)
            gloss_in = []
            if result == -1:
                # If not, just split up the glosses by commas.
                gloss_out = string.split(inside, ',')
            else:
```

```
            # Otherwise, get the stuff inside and surrounding the
            # inner square brackets.
            result = inrsqbraks.match(inside)
            if result == -1:
                print 'Malformed gloss in line', lineno
                sys.exit(1)
            left, cntr, right = inrsqbraks.group(1,2,3)

            # Split each part up by commas.  gloss_in will be the
            # stuff inside the brackets, and gloss_out will be the
            # stuff outside of them.
            gloss_l = string.split(left, ',')
            gloss_r = string.split(right, ',')
            gloss_in = string.split(cntr, ',')
            gloss_out = gloss_l
            gloss_out[len(gloss_out):len(gloss_out)] = gloss_r

        # Make a list of all the glosses for the current word.
        glos_l = []

        # Add the outer glosses.
        for gel in gloss_out:
            if gel != '':
                glos_l.append(gel)

        # Add the inner glosses, marking them with the first
        # letter 'O', for 'Object'.
        for gel in gloss_in:
            if gel != '':
                glos_l.append('O'+gel)

        # Actually add the rest of the glosses to the paradigm.
        self.paradigms.add_gloss(el[0], glos_l)

def parse_morphemes(self, file):
    """Read *file*, which should be a group of lines, each line
    containing a phonemic description of a word (not broken into
    morphemes, but with all obscuring phonological rules undone)
    and a gloss for the word.  The form of the gloss is as follows:
    [outer_feature*,[inner_feature*],outer_feature*], where
    outer_feature can either be a word in all capital letters
    (represents a semantic entity such as DOG) or a feature, which
    can be more or less anything except a word in all capital letters,
    although it cannot contain square brackets, commas, or white space.
    An inner_feature can only be a feature.  The outer_features
    represent either features of the subject of a verb or features
    of the word as a whole.  The inner_features represent features
    of the object of a verb.
    """

    # Open the file.
    try:
        f = open(file, 'r')
    except IOError:
        print 'Error opening file.\n'
        sys.exit(1)

    # Read the file and assemble a list of (word, gloss) pairs.
    try:
        word_list = []
        ln = f.readline()
        while ln != '':
            words = string.split(ln)
            if len(words) > 1:
                word_list.append((words[0],words[1]))
            ln = f.readline()
    except EOFError:
        print 'End.'

    # Parse the glosses into the corresponding paradigms
    self.parse_glosses(word_list)

    # Find the morphemes in each paradigm.
    self.paradigms.display()
    self.paradigms.find_morphemes()

    # Close the file.
    f.close()
```