TR-IT-0084
CO-OC: Semi-automatic Production of Resources for Tracking Morphological and Semantic Co-occurrences in Spontaneous Dialogues

Mark Seligman

December, 1994

Abstract

This document presents a tentative set of Communicative Acts (CAs) which has been used to label spontaneous dialogues in both English and Japanese. A Communicative Act is a communicative goal or aim which (according to native judgments) can be expressed in language L by a distinctive set of conventional cue patterns in specified discourse contexts. Communicative Acts are thus similar to speech acts, and similar to the pragmatic categories often called IFTs (illocutionary force types) at ATR. However, we restrict our attention to communicative goals which can be explicitly expressed via conventional surface cue patterns, thus excluding goals which are expressed using one-time-only combinations, goals which are expressed only implicitly, or goals which can only be defined in terms of relations between utterances. We describe methods of discovering and revising CAs which depend on native judgments concerning essential equivalence of meanings and functions of cue patterns in context. While these judgments are subjective, they concern shared conventions regarding objectively observable objects – the cue patterns and contexts. Thus a consensus can be expected to emerge during repeated revision. In this important respect, the methodology is data-driven or corpus-based. The present study also emphasizes comparison of CAs in English and Japanese. We find that most of our proposed CAs are valid for both English and Japanese: only two out of 27 CAs seem to be monolingual for our corpus. We begin by introducing CAs and briefly describing the background, goals, and status of our research. In later sections, we discuss our methodology in greater depth; we list our current Communicative Acts, with descriptive glosses, representative sets of surface patterns, examples, and other information; we present two labeled dialogues in English and two in Japanese; and finally, we provide an Appendix describing work in progress.

ATR音声翻訳通信研究所
ATR Interpreting Telecommunications Research Laboratories

Abstract

In the processing of spontaneous language, information concerning discourse-level co-occurrences of words or morphemes – relatively long-term predictions on the scale of several utterances – can reduce perplexity in speech recognition, aid disambiguation (by providing a measure of relatedness among elements which can help to score competing analyses), and help determine the boundaries of long-term topics in a discourse (by providing a coherence metric whose minima indicate topic transitions). However, flexible facilites for collecting, accessing, and viewing such co-occurrence information have been lacking. Further, because suitable corpora are scarce, sparse data is a serious problem, and smoothing is necessary. Semantic smoothing, in which one tracks co-occurrences of semantic tokens associated with words/morphs rather than co-occurrences of the words/morphs themselves, is a promising technique. This report describes a new set of facilities for tracking co-occurrences among morphs and/or their associated thesaurus codes and demonstrates the usability of the information obtained, stressing the problems and benefits of semantic smoothing. The benefits appear especially in the toolset's ability to retrieve reasonable semantically-mediated associations for morphs not in the original corpus (morphologically-tagged transcripts of 16 spontaneous Japanese dialogues concerning direction-finding and hotel arrangements). The tools support the use of various measures of association strength among morphs and their semantic labels, including standard calculations of conditional probability and mutual information (with optional use of standard statistical smoothing techniques). Very modest linguistic work would enable use of alternative thesauri or transcription formats. Potential uses for the new facilities are discussed in conclusion. Finally, a user's guide to CO-OC facilities and usage is provided.

1 Introduction

In the processing of spontaneous language, the need for predictions at the morphological or lexical level is clear. For bottom-up parsing based on phones or syllables, the number of lexical candidates is explosive. It is crucial to predict which morphological or lexical items are likely so that candidates can be weighted appropriately.

N-grams provide such predictions only at very short ranges. To support bottom-up parsing of noisy material containing gaps and fragments, longer-rang predictions are needed as well. Such predictions should have the advantage of being stronger than very short-range predictions, since predicting what will come "soon" is in general easier than predicting what will come next; and they should require less data, since examples of occurrence "soon" will be found more often in a corpus than examples of consecutive occurrence.

While stochastic grammars [Black 1993] can provide somewhat longer-range predictions than n-grams, they predict only within utterances, while our interest extends to predictions on the scale of several utterances. For help in predicting on this discourse scale, we might think of discourse-oriented mechanisms such as centering and global focusing models [Grosz 1986] [Walker 1992]; but in fact these are not designed to predict the lexical items that will be seen a bit later in the dialogue. Rather, as models of memory and attention, aiming primarily to clarify problems of reference, they provide records of the items which have already been seen and indications of their respective salience, but do not tell us what related items we should predict.

Church [Church 1990] has proposed an investigation of associations beyond the n-gram range which is closer to our present purpose; but the associations remained relatively short-range (on the order of 5 words).

The present proposal, then, is to permit the flexible definition of windows in a transcribed corpus within which co-occurrences of morphological or lexical elements can be examined. We describe CO-OC, a set of facilites for collecting, accessing, and viewing such co-occurrence information. Because suitable corpora are scarce and sparse data is a serious problem, smoothing is necessary. And so, in addition to standard statistical smoothing procedures, we propose techniques for semantic smoothing, in which one tracks co-occurrences of semantic tokens associated with words/morphs rather than co-occurrences of the words/morphs themselves. Their benefits appear especially in the possibility of retrieving reasonable semantically-mediated associations for morphs not in an original corpus. The tools we will describe support the use of various measures of association strength among morphs and their semantic labels, including standard calculations of conditional probability and mutual information (with optional use of standard statistical smoothing techniques). Potential uses for the new facilities are discussed in conclusion.

2 Segments and Windows

We first permit the investigator to define minimal segments within the corpus: these may be utterances, sections bounded by pauses or significant morphemes such as conjunctions, hesitations, postpositions, etc. Windows composed of several successive minimal segments can then be recognized: let Si be the current segment and N be the number of additional segments in the window as it extends to the right. N = 2 would, for instance, give a window three segments long with Si as its first segment. Then if a given word or morpheme M1 occurs (at least once) in the initial segment, Si, we attempt to predict the other words or morphemes which will co-occur (at least once) anywhere in the window.

Specifically, a conditional probability Q can be defined as follows:

```
Q(M1, M2) = P(M2 element of Si U Si+1 U Si+2 ... Si+n | M1 element of Si)

M1, M2 = morphemes
S1, S2 ... = segments
n = width of window in segments
```

Q is the conditional probability that M2 is an element of the union of segment Si, Si+1, Si+2, and so on up to Si+n, given that M1 is an element of Si.

If n = 0, the window is a single segment. In this case, Q indicates the probability that M2 co-occurs in segment Si, given that M1 occurs there. If n is greater than or equal to 1, Q is the probability that M2 will be found in any of the window's several segments. (Thus, while Q usually predicts M2 later in a window, M2 may sometimes precede M1 if it occurs in window-initial segment Si.)

Both the segment definition and the the number of segments in a window can be adjusted to vary the range over which co-occurrence predictions are attempted. Very long-range predictions will be strong but not informative; very short-range predictions (such as those given by consecutive n-grams) will be weak, but a great deal of information is provided if they come true. For maximum usefulness to speech recognition and analysis, we will hope to optimize through experimentation the combination of prediction strength and informativeness.

3 Corpus and Early Experiments

For initial experiments, we used a morphologically-tagged corpus of 16 spontaneous Japanese dialogues concerning direction-finding and hotel arrangments [Loken-Kim 1993]. We collected common-noun/common-noun, common-noun/verb, verb/common-noun, and verb/verb conditional probabilities in a three-segment window (n = 2). Conditional probability Q is computed among all morph pairs for these classes and stored to a database; pairs scoring below a threshold (0.1 for the initial experiments) were discarded. We also compute and store the mutual information for each morph pair, using the standard definition as [Fano 1961].

Fast queries of the database are then enabled. A central function is GET-MORPH-WINDOW-MATES, which provides all the window mates for a specified morph which belong to a specified class and have scores above a specified threshold for the specified co-occurrence measure (conditional probability or mutual information).

The intent is to use such queries in real time to support bottom-up, island-driven speech recognition and analysis. To support the establishment of island centers for such parsing, we also collect information on each corpus morph in isolation: its hit count and the segments it appears in, its unigram probability and probability of appearance in a segment, its probability of appearance in any given segment, etc. Once island hypotheses have been established based on this foundation, co-occurrence predictions will come into play for island extension. Global information concerning morphs is also recorded, showing that our present 16-dialogue corpus, in which a minimal segment has been defined as a single utterance, contains 1743 segments; is 19250 morphs long; has 949 different morphs; and has a morph unigram entropy of 6.8982.

4 Semantic Smoothing

It was suggested that sparse data should be somewhat less problematic for long-range than for short-range predictions. Still, there is never quite enough data; so abstraction, or smoothing, of the data will remain desirable. As a statistical smoothing measure, we presently use the standard

1+ technique (adding one to each event count) [Nadas 1985] for both conditional probability and mutual information.

In addition, however, we enable semantic smoothing in an innovative way. Thesaurus categories – cats for short – are sought for each corpus morph (and stored in a corpus-specific customized thesaurus for fast access). The common-noun *eki* (station), for instance, has among others the cat label "725a" (representing a semantic class of posts-or-stations) in the standard Kadokawa Japanese thesaurus [Ohno 1981]. (We also experimented with a different thesaurus customized for ATR use. It initially appeared helpful in providing syntactic information in entries, thus minimizing access ambiguity; but as its general-purpose coverage was less extensive it proved unsuitable for our current experiments. It is significant, however, that CO-OC can accommodate a new thesaurus format with about a day's labor.)

Equipped with such information, we can study the co-occurrence within windows of cats as well as morphs. For example, using n = 2, GET-CAT-WINDOW-MATES finds 36 cats co-occurring with "725a", one of the cats associated with *eki* (station), with a conditional probability Q greater than 0.10, including "459a" (*sewa*, taking-care-of or looking-after), "216a" (*henkou*, transfer), and "315b" (*ori*, getting-off). Since we have prepared an indexed reverse thesaurus for our corpus, we can quickly find the corpus morphs which have these cat labels, respectively *miru*, "look", *mieru*, "can see, visible"; *magaru*, "turn"; and *oriru*, "get off". The resulting morphs are related to the input morph *eki* via semantic rather than morph-specific co-occurrence. They thus form a broader, smoothed group.

This semantic smoothing procedure – morph to related cats, cats to co-occurring category window-mates, cats to related morphs – has been encapsulated in the function GET-MORPH-WINDOW-MATES-VIA-CATS. It permits filtering, so that morphs are output only if they belong to a desired morphological class and are mediated by cats whose co-occurrence likelihood is above a specified threshold.

Thesaurus categories are normally arranged in a type hierarchy. In the Kadokawa thesaurus, there are four levels of specificity: "725a" (posts-or-stations), mentioned above, belongs to a more general category "725" (stations-and-harbors), which in turn belongs to "72" (institutions), which belongs to "7" (society). Accordlingly, we need not restrict co-occurrence investigation to cats at the level given by the thesaurus. Instead, knowing that "725a" occurred in a segment Si, we can infer that all of its ancestor cats occurred there as well; and can seek and record semantic co-occurrences at every level of specificity. This has been done; and GET-MORPH-WINDOW-MATES-VIA-CATS has a parameter permitting specification of the desired level of semantic smoothing. The more abstract the level of smoothing, the broader the resulting group of semantically-mediated morpheme co-occurrences.

The most desirable level for semantic smoothing is a matter for future experimentation. However, we can anticipate a general preference for the most specific predictions available: we would resort to semantic smoothing only when no morph-specific co-occurrences could be predicted at a certain threshold, and would resort to more abstract semantic smoothing only when more specific smoothing failed. Thus we provide a function GET-MORPH-WINDOW-MATES-MOST-SPECIFIC with this behavior. Its value for robustness appears especially in cases when the input is a morph which did not occur in the training corpus. Without semantic-smoothing-in-case-of-need, the attempt to make co-occurrence predictions would certainly fail; but with this possibility, reasonable predictions can often be made. An entry for the new morph is sought dynamically in the relevant thesaurus; any cats thus found are checked for likely co-occurring cats; and morphs associated with these cats in the training corpus can be delivered as output. For instance, *kuruma*, "car, auto", does not appear in our corpus. However, the Kadokawa thesaurus does list this morph with codes "997" (vehicles) and "985" (wheels), yielding a wide range of associated verbs from our corpus, including *iku*, "go", *tuku*, "arrive", and 44 others; and of common-nouns, including *shibasu*, "city bus", *ikikata* "way to go, means of transportation", and 52 others. For each morph retrieved in

this way, the conditional probability of the mediating cat co-occurrence can be recovered.

5 Knowledge Resources

The data resources we have described can be viewed as a knowledge base containing both type and non-type relations. Type relations appear in the grouping of morphs according to their morphological classes (in our present investigation, common-nouns and verbs), and in the arrangement of cats in a subtree of the original thesaurus hierarchy. Non-type relations include the many-to-many links among morphs and cats and the co-occurrence relations between morphs and between cats. The co-occurrence relations, in particular, form a network whose uses will be discussed below.

The production of the knowledge base is automatic except for a very modest amount of linguistic work to be described just below.

To facilitate viewing of these various objects and relationships, we have provided facilities for downloading the information into a commercially-available object-oriented expert system shell, KEE(TM), chosen for its interface-building capabilities. The abovementioned type relations are apparent, with morphs and cats appearing as browsexable objects whose slots can be examined for statistical information or non-type relational information. Simple improvements would allow representation of non-type relations as labeled lines. Co-occurrence relations could easily be depicted as lines whose length represented the conditional probability, so that mutually-predicting items would visibly cluster.

6 Resolving Morphs.to.cats Mapping Problems

During thesaurus look-up for semantic smoothing, problems of failure or ambiguity sometimes occurred.

Often failures could be corrected by regularizing morphs to dictionary forms. Our verb morphs, for instance, appeared in transcripts as stems (e.g. *ka*) and required conversion to dictionary forms like *kaku*, "write". Since our corpus contained only 114 verbs, a handmade mapping file meeting this need could be quickly prepared; in a larger corpus, more spohisticated morphological processing might be warranted. After regularization there were five remaining verb failures, reflecting differences in analysis of derived forms between the transcriber and the thesaurus format, giving a verb coverage of almost 96 percent. Similarly, a number of common-noun failures stemmed from the inclusion of compounds in the transcript; handmade mappings to head nouns and other similar adjustments gave a final coverage of better than 92 percent of common-nouns. It is important that this mapping, made with the help of a native Japanese speaker in less than a person-day, was the only linguistic work necessary to permit the otherwise automatic compilation of the knowledge resources described throughout.

In addition to thesaurus lookup failures, lookup ambiguity of two types occurred. First, certain nouns transcribed using Japanese syllable script (hiragana) rather than kanji characters corresponded to multiple thesaurus entries. Such ambiguity, like lookup failures, could often be corrected using handmade mappings, in this case specifying a kanji character with a unique entry. After mapping correction, only common-nouns with multiple entries remained in our corpus. Second, morph-to-cat ambiguity routinely occured because each entry normally yields several cat codes, usually representing different senses of a given morph. In the future, customization of the thesaurus might be tried as a remedy, but for now we simply tolerate a resulting degradation of accuracy in our predictions of cat (and cat-mediated morph) co-occurrences.

7 Evaluation

We are presently reporting the implementation of facilities intended to enable many experiments concerning morphological and morpho-semantic co-occurrence; the experiments themselves remain for the future. Nevertheless, some indication of the usablility of the data is in order.

Tools have been provided for comparing two corpora with respect to any of the fields in the records relating to morphs, morph co-occurrences, cats, or cat co-occurrences. Using these, we treated 15 of our dialogues as a training corpus, and the one remaining dialogue as a test corpus. We compared the two corpora in terms of unigram probabilities for morphs, and in terms of conditional probabilities for morph co-occurrences. (In both cases, statistically unsmoothed scores were used for simplicity of interpretation.) (As already mentioned, for the studies described here we used utterances as minimal segments and sought co-occurrences in windows of three minimal segments (n = 2). Recall too that co-occurrences were sought for all combinations of common-nouns and verbs.)

Considering all morphological classes, we found 898 different morphs in the training corpus and 365 in the test. 314 morphs were found in both corpora; so our training corpus of 15 dialogues covered 314 out of 365 morphs in the test dialogue, or about 86 percent. Table 1 compares the corpora for 25 shared common-nouns, and Table 2 compares 25 shared verbs. Morphs are listed in order of least difference between corpora.

Table 1: Training and test corpora compared: 25 shared common-nouns

あと ato	4.0E-4	4.0E-4	0.0	7	1	8
近辺 kinpen	4.0E-4	4.0E-4	0.0	7	1	8
とこ toko	4.0E-4	4.0E-4	0.0	6	1	7
自宅 jitaku	4.0E-4	4.0E-4	0.0	6	1	7
後 ato	4.0E-4	4.0E-4	0.0	6	1	7
何か nanika	4.0E-4	4.0E-4	0.0	6	1	7
出口 deguchi	0.0024	0.0025	1.0E-4	41	6	47
市バス shibasu	7.0E-4	8.0E-4	1.0E-4	11	2	13
方面 houmen	5.0E-4	4.0E-4	1.0E-4	8	1	9
画面 gamen	3.0E-4	4.0E-4	1.0E-4	5	1	6
辺 atari	3.0E-4	4.0E-4	1.0E-4	5	1	6
あたり atari	3.0E-4	4.0E-4	1.0E-4	5	1	6
今回 konkai	3.0E-4	4.0E-4	1.0E-4	5	1	6
行 i	0.0023 0	.0025	2.0E-4	39	6	45
バス basu	0.0018	0.0016	2.0E-4	31	4	35
左手 hidarite	0.001	0.0012	2.0E-4	16	3	19
建物 tatemono	0.001	8.0E-4	2.0E-4	16	2	18
上 ue	2.0E-4	4.0E-4	2.0E-4	4	1	5
夕方 yuugata	2.0E-4	4.0E-4	2.0E-4	4	1	5
フロント furonto	2.0E-4	4.0E-4	2.0E-4	4	1	5
斜め naname	2.0E-4	4.0E-4	2.0E-4	4	1	5
位置 ichi	2.0E-4	4.0E-4	2.0E-4	3	1	4
下 shita	2.0E-4	4.0E-4	2.0E-4	3	1	4
形 katachi	2.0E-4	4.0E-4	2.0E-4	3	1	4
T T	2.0E-4	4.0E-4	2.0E-4	3	1	4

As to morph co-occurrences, we found 5162 co-occurrence pairs above a conditional probability threshold of 0.10 in the training corpus and 1552 in the test. Since 509 pairs occurred in both corpora, the training corpus covered 509 out of 1552, or 33 percent, of the test corpus. That is, one third of the morph co-occurrences with conditional probabilities above 0.10 in the test corpus were anticipated by the training corpus.

Table 2: Training and test corpora compared: 25 shared verbs

乗り継 noritsu	4.0E-4	4.0E-4	0.0	7	1	8
かま kama	4.0E-4	4.0E-4	0.0	6	1	7
あ a	0.0026	0.0025	1.0E-4	43	6	49
歩 aru	0.0017	0.0016	1.0E-4	29	4	33
し shi	7.0E-4	8.0E-4	1.0E-4	11	2	13
降り ori	0.0014	0.0012	2.0E-4	24	3	27
書 ka	0.0014	0.0012	2.0E-4	24	3	27
教え oshie	0.0014	0.0012	2.0E-4	24	3	27
離れ hanare	2.0E-4	4.0E-4	2.0E-4	4	1	5
目立 meda	2.0E-4	4.0E-4	2.0E-4	3	1	4
願 nega	7.0E-4	4.0E-4	3.0E-4	12	1	13
言 i	7.0E-4	4.0E-4	3.0E-4	11	1	12
取 to	5.0E-4	8.0E-4	3.0E-4	9	2	11
行け ike	1.0E-4	4.0E-4	3.0E-4	2	1	3
分れ wakare	1.0E-4	4.0E-4	3.0E-4	2	1	3
か ka	1.0E-4	4.0E-4	3.0E-4	1	1	2
打 u	1.0E-4	4.0E-4	3.0E-4	1	1	2
違 chiga	1.0E-4	4.0E-4	3.0E-4	1	1	2
変わ kawa	1.0E-4	4.0E-4	3.0E-4	1	1	2
す su	1.0E-4	4.0E-4	3.0E-4	1	1	2
見 mi	4.0E-4	8.0E-4	4.0E-4	7	2	9
描 ka	4.0E-4	8.0E-4	4.0E-4	6	2	8
な na	0.002	0.0025	5.0E-4	34	6	40
戻 modo	7.0E-4	0.0012	5.0E-4	11	3	14
思 omo	0.0023	0.0029	6.0E-4	39	7	46

This coverage seems respectable, considering that the training corpus was small and that neither statistical nor semantic smoothing was used. More important than coverage, however, is the presence of numerous pairs for which good co-occurrence predictions were obtained. Such predictions differ from those made using n-grams in that they need not be chained, and thus need not cover the input to be useful: if consistently good co-occurrence predictions can be recognized, they can be exploited selectively.

Table 3 shows pair comparisons for the 35 pairs which occured most often, taking the sum of counts in both corpora. Pairs are ordered by least difference between corpora, so that the best predictions appear first.

Comparable figures were obtained for cats and cat co-occurrences.

8 Discussion

The primary purpose of the co-occurrence information under discussion is prediction at the morphological or lexical level in the service of speech recognition.

The conditional probabilities fetched from the database can be used in two different ways. They can be used to filter speech and analysis hypotheses which have already been made; or they can be used to dynamically weight or select the rules which are used for speech recognition and analysis, in order to dynamically influence the treatment of hypotheses. In this case, for instance, speech recognition does not need to evaluate every morpheme in the lexicon, but can concentrate on morphemes whose predicted probability passes a certain threshold.

Table 3: The 35 most frequent pairs over both corpora

路 michi	C-NOUN	字 ji	C-NOUN	1.0	1.0	0.0	21	1	22
乗り継 noritsu	VERB	電車 densha	C-NOUN	1.0	1.0	0.0	7	1	8
目 me	C-NOUN	前 mae	C-NOUN	1.0	1.0	0.0	5	2	7
自宅 jitaku	C-NOUN	番号 bango	C-NOUN	1.0	1.0	0.0	6	1	7
わかり wakari	C-NOUN	な na	VERB	1.0	1.0	0.0	4	2	6
画面 gamen	C-NOUN	地図 chizu	C-NOUN	1.0	1.0	0.0	5	1	6
フロント furonto	C-NOUN	よう you	C-NOUN	1.0	1.0	0.0	4	1	5
フロント furonto	C-NOUN	名前 namae	C-NOUN	1.0	1.0	0.0	4	1	5
目立 meda	VERB	わか waka	VERB	1.0	1.0	0.0	3	1	4
目立 meda	VERB	建物 tatemono	C-NOUN	1.0	1.0	0.0	3	1	4
T	C-NOUN	見え mie	VERB	1.0	1.0	0.0	3	1	4
T	C-NOUN	曲が maga	VERB	1.0	1.0	0.0	3	1	4
T	C-NOUN	歩 aru	VERB	1.0	1.0	0.0	3	1	4
T	C-NOUN	路 michi	C-NOUN	1.0	1.0	0.0	3	1	4
T	C-NOUN	左 hidari	C-NOUN	1.0	1.0	0.0	3	1	4
T	C-NOUN	字 ji	C-NOUN	1.0	1.0	0.0	3	1	4
T	C-NOUN	方 hou	C-NOUN	1.0	1.0	0.0	3	1	4
変え kae	VERB	地図 chizu	C-NOUN	1.0	1.0	0.0	1	2	3
行け ike	VERB	行 i	VERB	1.0	1.0	0.0	2	1	3
コンコース	C-NOUN	出口 deguchi	C-NOUN	1.0	1.0	0.0	1	1	2
いくつ ikutsu	C-NOUN	バス停 basutei	C-NOUN	1.0	1.0	0.0	1	1	2
学会 gakkai	C-NOUN	教え oshie	VERB	1.0	1.0	0.0	1	1	2
学会 gakkai	C-NOUN	場所 basho	C-NOUN	1.0	1.0	0.0	1	1	2
夜 yoru	C-NOUN	シングル	C-NOUN	1.0	1.0	0.0	1	1	2
待 ma	VERB	書 ka	VERB	0.2024	0.202	4.0E-4	25	5	30
行 i	VERB	出 de	VERB	0.1683	0.1688	5.0E-4	77	6	83
名前 namae	C-NOUN	よう you	C-NOUN	0.2537	0.2532	5.0E-4	20	4	24
教え oshie	VERB	方 hou	C-NOUN	0.3354	0.3361	7.0E-4	24	3	27
方 hou	C-NOUN	ござ goza	VERB	0.198	0.2	0.002	131	10	141
出口 deguchi	C-NOUN	な na	VERB	0.2	0.202	0.002	35	5	40
出口 deguchi	C-NOUN	電車 densha	C-NOUN	0.2	0.202	0.002	35	5	40
よう you	C-NOUN	フロント furonto	C-NOUN	0.3333	0.3361	0.0028	12	3	15
出 de	VERB	正面 shoumen	C-NOUN	0.1749	0.1688	0.0061	69	6	75
出 de	VERB	行 i	C-NOUN	0.1749	0.1688	0.0061	69	6	75
左 hidari	C-NOUN	歩 aru	VERB	0.5	0.5063	0.0063	26	2	28

Two secondary uses of co-occurrence information should also become possible: (1) to aid future studies of disambiguation; (2) to aid in the recognition of the boundaries of "discourse segment topics".

Regarding disambiguation, several studies have proposed using closeness in semantic networks as a mechanism for resolving both lexical and structural ambiguity. (Closeness can be measured using the number of links, their strengths, or both.)

Consider lexical ambiguity first. Suppose the English word *bank* has been found in the current utterance. If the words in recent history include *money, transfer, deposit*, etc., the meaning "saving and lending institution" will be preferred over the meaning "edge of river", because the first concept is closer to the concepts of the recent words. Now consider structural ambiguity. In *I saw the man in the park with the telescope*, should we attach with the telescope to *saw*, or *man*, or *park*? The first is preferred, because the paths between the concepts "telescope" and "see" are shorter than those between "telescope" and "man" or "park". It may well be possible to apply a network of co-occurrence relations for disambiguation in this way.

Consider next the recognition of topic boundaries within a discourse, sometimes required in attempts to resolve pronoun referents over relatively long distances. Several studies have suggested the use of cohesion metrics for this boundary recognition: given a definition of cohesion between adjacent windows of a discourse, it becomes possible to seeks places in a discourse where the cohesion is lowest. These are hypothesized as topic boundaries. Co-occurrence relations can provide such a cohesion metric, thus presenting an alternative to the boundary discovery methods suggested in e.g. [Morris 1991], [Hearst 1994], [Nomoto 1994], and [Kozima 1994].

9 Files and Programs

This section provides an introduction to CO-OC files and structures.

Processing begins with EMMI TRANSCRIPT files without suffixes, e.g.

```
1a-AMWX
```

The content appears as follows:

```
A：はい／国際会議事務局／で／す
80/11/100/30
MW：［／あ／］／すいません／［／あのー／］／きょう／の／翻訳／（／電／）
／電話通信国際シンポジウム／の／会場／へ／行／き／た／い／ん／で／／けど
90/80/90/14/111/11/11/118/14/111/21/30/100/30/112/100/30/116
...
```

9.1 Conversion of Transcripts to .MORPHS Format

The first stage of corpus pre-processing is to convert transcript files into a one-morph-per-line format (similar to the format of the ATR Dialogue Database) for computational convenience and readability.

The result is a .MORPHS file for each transcript file, e.g.

```
1a-AMWX.morphs
```

The transformed content looks like this:

```
(はい 感動詞)
(国際会議事務局 固有名詞)
(で 助動詞)
(す 語尾)
(END-OF-TURN)
(あ 間投詞)
(すいません 感動詞)
(あのー 間投詞)
(きょう 普通名詞)
(の 格助詞)
(翻訳 固有名詞)
((電) NIL)
```

(電話通信国際シンポジウム 固有名詞)
(の 連体助詞)
(会場 普通名詞)
(へ 格助詞)
(行 本動詞)
(き 語尾)
(た 助動詞)
(い 語尾)
(ん 準体助詞)
(で 助動詞)
(す 語尾)
(けど 接続助詞)
(END-OF-TURN)

During reformatting, the morphology number codes which ap- pear in the input can optionally be rewritten as Japanese or English symbols for readability.[1] The global variable *pos-table* defines all correspondences.

9.2 Corpus Segmentation

The corpus, which may contain several .MORPHS files, is then *segmented* using a set of boundary markers (see further below), giving a file for the entire corpus like

```
1a.8b.segments
```

The simplest procedure is to break the corpus into turns, using only END-OF-TURN (supplied during the conversion to .morphs format) as a marker. However, if the transcript includes pauses, these can also be used, so that the segments are "pause units" rather than turns; or specified morphemes, e.g. conjunctions, hesita- tions, postpositions like *wa*, etc. could also be used.

9.3 Corpus Analysis

Once segmentation is complete, analysis can begin. The CO-OC universe contains MORPHS (morphemes) and their associated CATS (thesaurus categories). We want information about both. Information about morphs is kept in MORPHRECS (morph records), and information about cats is stored in CATRECS.

We also want information about the *co-occurrences* of morphs and of cats. For this purpose, we maintain COOCRECS (co-occurrence records). There are MORPH-COOCRECS and CAT-COOCRECS.

We have mentioned four kinds of records: MORPHRECS, CATRECS, MORPH-COOCRECS, and CAT-COOCRECS. See the appropriate sections of Appendix 1 for definitions of these structures. In each case, ordered list structures have been used for maximum simplicity, avoiding the use of DEFSTRUCT, etc.

Some Essential Files Each kind of record is stored in two kinds of files: an .INDEXED file for fast access, and a .SORTED file for ease of sorting, browsing, KEE conversion, etc. The

[1] English labels have been used for the preliminary experiments described throughout this report. Time did not permit full testing of the programs with Japanese or numbered labeling.

.INDEXED file has a tree or discrimination-network structure, while the .SORTED file is a simple list.

Tree format for morphrecs.indexed:

```
((<pos>
  (<list of bare morphs>)
  (<morphrec><morphrec> ...))
 (<pos>
  (<list of bare morphs>)
  (<morphrec><morphrec> ...))
 (<pos>
  (<list of bare morphs>)
  (<morphrec><morphrec> ...))
 ...)
```

The catrecs.indexed tree is a D-net structure composed of NODES. See the NODE-STRUCTURE definition in Appendix 1, section II.D.

Morph-coocrecs.indexed and cat.coocrecs.indexed each have unique tree structures for fast access. See sample files for examples. (Note especially that the terminals of these trees are not complete coocrecs. Rather, when coocrecs are to be fetched, they are composed dynamically by combining items harvested during descent of the indexed tree with partial coocrecs at tree terminals.)

Now we have eight files for a given corpus. The EMMI corpus begins with dialogue 1a and ends with 8b. We use the notation 1a.8b to indicate an inclusive span of files. So for this corpus we will expect these eight output files (note the dot convention for filenames – and note that dashes are used instead within Lisp):

```
1a.8b.morphrecs.indexed
1a.8b.morphrecs.sorted
1a.8b.catrecs.indexed
1a.8b.catrecs.sorted
1a.8b.morph.coocrecs.indexed
1a.8b.morph.coocrecs.sorted
1a.8b.cat.coocrecs.indexed
1a.8b.cat.coocrecs.sorted
```

These files contain information about individual MORPHRECS, CATRECS, etc. (See Appendix 1 concerning the format of these records.) But we also want global information about all of the morphrecs in the corpus. For this purpose, we make a MORPHS.GLOBAL file, presently containing

```
all-segments-count
discourse-final-segments
corpus-length-in-morphemes
number-of-different-morphemes
morph-unigram-entropy
```

Intermapping Morphs and Cats Morphs and cats are intermapped. We need to be able to quickly find which cats belong to a morph or vice versa. So we maintain indexed files for a corpus, e.g.

```
1a.8b.morphs.to.cats
1a.8b.cats.to.morphs
```

The morphs.to.cats structure is identical to that of morphrecs.indexed, while that of cats.to.morphs is identical to the structure of catrecs.indexed.

For linguistic work when tuning a thesaurus to a training corpus, we need to know which morphs failed to find cats, or found multiple (and thus ambiguous) entries. So we have e.g.

```
1a.8b.morphs.to.cats.global
```

which now contains this information for each investigated part of speech:

```
morphrecs-length
failures
failure-ratio
morphrecs-without-entries
morphrecs-with-multiple-entries
```

The abovementioned tuning is carried out, as explained in previous sections, using .DICTIONARY.FORMS files. As we presently use the FULL version of the Kadokawa thesaurus, we have

```
full.common.noun.dictionary.forms
full.verb.dictionary.forms
```

Comparing Records in Two Corpora Once this information has been generated for a given corpus, we want to *compare* MORPHRECS, CATRECS, MORPH-COOCRECS, and CAT-COOCRECS for various corpora, especially for a training and a test corpus. So we have various .COMPARISON files, e.g.

```
1a.8a.vs.8b.morphrecs.comparison
1a.8a.vs.8b.morph.coocrecs.comparison
1a.8a.vs.8b.catrecs.comparison
1a.8a.vs.8b.cat.coocrecs.comparison
```

In the .MORPHRECS.COMPARISON file we have for example

```
;;;~/survey/jap/data/1a.8a.morphrecs.sorted and ~/survey/jap/data/8b.morphrecs.sorted
;;;compared in terms of MORPH-UNIGRAM-PROBABILITY

(relative-entropy 0.2415)

(average-MORPH-UNIGRAM-PROBABILITY-corpus-1 0.0026)

(average-MORPH-UNIGRAM-PROBABILITY-corpus-2 0.0027)

(average-MORPH-UNIGRAM-PROBABILITY-difference  8.0E-4)

(in-both-morphs-count 314)
```

```
(difference-records-corpus-1-order ((す INFLECTIONAL-ENDING 0.0629 0.0682 0.0053 1058 166 1224)
(で AUX-VERB 0.0468 0.0509 0.0041 787 124 911) ...))

(in-corpus-1-but-not-corpus-2-count 584)

(in-corpus-1-but-not-corpus-2 ((はあ FILLED-PAUSE) (という ADNOMINAL-PART) ...))

(in-corpus-2-but-not-corpus-1-count 51)

(in-corpus-2-but-not-corpus-1 ((前後 COMMON-NOUN) (用意 SAHEN-NOUN) ...))
```

The .COMPARISON files introduce a new kind of record, a DIFFREC (difference record). There are MORPHREC-DIFFRECS, CATREC-DIFFRECS, MORPH-COOCREC-DIFFRECS, and CAT-COOCREC-DIFFRECS.

9.4 KEE Interface

Sample code has also been provided for downloading morphrecs and catrecs to KEE, an object-oriented shell with excellent interface-building capabilities. So far it has been necessary to use only a half-dozen simple KEE functions. Some possibilities for future interface experimentation were mentioned in a previous section. KEE is used only for viewing CO-OC objects and relations; no processing is carried out within KEE.

Note that a separate file, kee.lisp has been prepared for use within KEE in order to prompt object creation. It is included here as Appendix 2.

This completes our overview of the files and datastructures used in CO-OC processing. The next section surveys basic program usage.

10 CO-OC Program Usage

We first survey the top-level control functions used to create CO-OC knowledge bases. We then look at high-level data access functions.

10.1 Building KBs

The highest-level KB-building function is HANDLE-CORPORA-AND-COMPARISONS. It attempts to automate the entire chain of events described above – it handles two corpora and then compares them, assuming only that .DICTIONARY.FORMS files have been prepared for the relevant thesaurus to minimize thesaurus access problems. (These can be omitted, however, if poor results can be tolerated.)

Here is the head of the function definition for HANDLE-CORPORA-AND-COMPARISONS showing its complete keyword argument set.

```
;;;Handles two corpora and compares them.
(defun handle-corpora-and-comparisons
    (n-range                          ;like (0 2)
     pos-1-list                       ;like (common-noun verb)
     pos-2-list                       ;like (common-noun verb)
```

```
directory ;like "~/survey/jap/data/"
transcript-file-prefixes-1 ;like ("1a-AMWX" "1b-AMWX")
transcript-file-prefixes-2 ;like ("2a-AMWX" "2b-AMWX")
corpus-prefix-1 ;like "1a.1b"
corpus-prefix-2 ;like "2a.2b"
morphs-to-cats-prefix ;if using pre-made files

&key
;;highest-level-control
(handle-corpus-1 t)
(handle-corpus-2 t)
(handle-morph-and-cat-comparisons t)

;;for handle-morph-and-cat-comparisons, if permitted
(handle-morph-comparisons t)
(handle-cat-comparisons t)

;;high-level control
(handle-morphs t)
(handle-cats t)

;;for handle-morphs, if permitted
(make-morphs-files t)
(segment-corpus t)
(make-morphrecs-indexed-and-sorted t)
(make-morph-coocrecs-indexed t)
(make-morph-coocrecs-sorted t)

(resort-by 'hit-count)
(separators '(END-OF-TURN))
(db-pos-display 'english)
(threshold 0.1)
(threshold-key 'conditional-probability)

;;for handle-cats, if permitted
(make-catrecs-sorted t)
(make-catrecs-indexed t)
(make-cat-coocrecs-sorted t)
(make-cat-coocrecs-indexed t)

;;for handle-mtc-and-ctm if permitted by used of
;;morphs-to-cats-prefix
(thesaurus 'thesaurus-full))
```

...

Examples are provided for all of the required arguments. See also sample program calls which are included in Appendix 1.

N-RANGE indicates the desired range of window widths for investigation: several values for N can be used in the construction of a single knowledge base.

POS-1-LIST and POS-2-LIST are lists of parts of speech whose co-occurrences are to be investigated. Each morph belonging to any pos in the first list will be tested against each morph

belonging to any pos in the second list.

DIRECTORY is the directory where transcript files are stored, and where completed knowledge base files will be sent.

TRANSCRIPT-FILE-PREFIXES-1 and -2 are the initial input for corpus-1 and corpus-2, respectively.

CORPUS-PREFIX-1 and -2 provide prefixes for automatic construction of appropriate file names for output.

MORPHS-TO-CATS-PREFIX is supplied only if thesaurus processing and intermapping of morphs and cats has already been performed, so that usable morphs.to.cats and cats.to.morphs file already exist. In this case, this argument provides a pointer to them. For instance, I performed this intermapping in advance for 1a through 8b; so I could supply a "1a.8b" value for this argument in order to save the time of re-mapping when experimenting with subsets of the full EMMI 16-dialogue corpus.

Many KEYWORD ARGUMENTS have been supplied to allow skipping of selected steps during debugging. For example, to bypass the re-handling of corpus-1 if it has already been handled, one can make a call using

```
:handle-corpus-1 nil
```

Note the THRESHOLD and THRESHOLD-KEY keyword arguments. The default threshold is 0.1, meaning that scores under that value are discarded at KB-build time; and the default threshold-key is CONDITIONAL-PROBABILITY, meaning that this is the field used for threshold judgments. (Other possible values are CONDITIONAL-PROBABILITY-SMOOTHED, MUTUAL-INFORMATION, and MUTUAL-INFORMATION-SMOOTHED).

The RESORT-BY keyword argument defaults to HIT-COUNT, meaning that CO-OC uses the value of this record field for its internal sorting when making both .SORTED and .INDEXED files for morphrecs, catrecs, etc. (.SORTED files are sorted from beginning to end, while .INDEXED files are sorted within indexed groupings.)

The THESAURUS keyword argument defaults to FULL. This is the Department 3 name for the full Kadokawa thesaurus. It is kept in

```
as29:/usr1/EBMT-DATA/THESAURUS/KADOKAWA/FULL
```

and its hierarchy is in

```
as29:/usr1/EBMT-DATA/THESAURUS/KADOKAWA/HIERARCHY
```

10.2 Data Access

The purpose of four important data access functions were introduced in a previous section:

```
get-morph-window-mates
get-cat-window-mates
get-morph-window-mates-via-cats
get-morphology-window-mates-most-specific
```

Examples of program use are included in Appendix 1, II.A.1. Here we display the heads of the program definitions and comment on selected arguments.

```
(defun GET-MORPH-WINDOW-MATES-WITH-FILES (n pos-1 lex-1 pos-2 directory corpus-prefix
       &key (threshold nil)
            (threshold-key 'conditional-probability)
            (filter-self t)
            (db-pos-display 'english)
            (return-morphs-only nil))
...)
```

We are seeking window-mates for the morph specified by POS-1 and LEX-1 with a window width of N, and specify that the mate must belong to POS-2. DIRECTORY and CORPUS-PREFIX jointly point to the relevant knowledge base files.

THRESHOLD and THRESHOLD-KEY have already been explained. See above regarding non-default values for THRESHOLD-KEY. FILTER-SELF is normally T, since a morph can always be its own window mate but we usually don't want to be told this. RETURN-MORPHS-ONLY should be T if a concise list of possible mates is wanted; otherwise a verbose list of complete morphrecs will be delivered.

```
(defun GET-CAT-WINDOW-MATES-WITH-FILES
    (n cat-1 directory corpus-prefix
       &key (threshold nil)
            (threshold-key 'conditional-probability)
            (filter-self t)
            (db-pos-display 'english)
            (return-cats-only nil))
...)
```

As for the -MORPHS- function above.

```
(defun GET-MORPH-WINDOW-MATES-VIA-CATS-WITH-FILES
    (n pos-1 lex-1 directory corpus-prefix morphs-to-cats-prefix
       &key
       (level nil)
       (threshold nil)
       (threshold-key 'conditional-probability)
       (filter-self t)
       (thesaurus 'thesaurus-full)
       (return-morphs-only nil)
       (cats nil))
...)
```

MORPHS-TO-CATS-PREFIX is needed as a pointer to a precomputed intermapping between morphs and cats.

LEVEL allows specification of the thesaurus level at which semantic smoothing will be tried. The default, NIL, means any level is permissible. CATS allows specification of the particular list of cats for which cat-coocrecs should be sought.

```
(defun get-morph-window-mates-most-specific-with-files
```

```
    (n pos-1 lex-1 pos-2 directory corpus-prefix morphs-to-cats-prefix
       &key
       (level nil)
       (threshold nil)
       (threshold-key 'conditional-probability)
       (filter-self t)
       (thesaurus 'thesaurus-full)
       (db-pos-display 'english)
       (return-morphs-only nil))
...)
```

As above.

REFERENCES

Black, E., R. Garside, and G. Leech. 1993. Statistically-driven Computer Grammars of English: The IBM/Lancaster Approach. Language and Computers: Studies in Practical Linguistics No. 8. Rodopi. Amsterdam, Atlanta GA. 1993.

Church, K. 1990. Word Association Norms, Mutual Information, and Lexicography. *Computational Linguistics,* 16, Number 1, March 1990.

Fano, R. 1961. Transmission of Information: A Statistical Theory of Communications. MIT Press, Cambridge, MA.

Grosz, B. and C. Sidner. 1986. Attention, Intensions, and the Structure of Discourse. *Computational Linguistics,* 12, pages 175-204.

Hearst, M. 1994. Multi-paragraph segmentation of expository text. In *Proceedings of the 32nd Annual Meeting of the Association for Computational Linguistics*, Las Cruces, June 27-30, 1994.

Kozima, H. and T. Furugori. 1994. Segmenting Narrative Text into Coherent Scenes. Literary and Linguistic Computing, Volume 9, Number 1.

Loken-Kim, K. and F. Yato. 1993. EMMI-ATR Environment for Multi-modal Interaction. ATR Interpreting Telephony Research Laboratories, ATR Technical Report TR-I-0118. Also in EACL-89.

Mitamura, M., E. Nyberg, 3rd, and J. Carbonell. 1993. Automated Corpus Analysis and the Acquisition of Large, Multi-lingual Knowledge Bases for MT. In *Proceedings of the Fifth International Conference on Theoretical and Methodological Issues in Machine Translation.* Kyoto, Japan, July 14-16, 1993.

Morris, J. and G. Hirst. 1991. Lexical cohesion computed by thesaural relations as an indicator of the structure of text. *Computational Linguistics,* 17, pages 21-48.

Nomoto, T. and Y. Nitta. 1994. A Grammatico-statistical Approach to Discourse Partitioning. In *Proceedings of COLING-94*, Aug. 5-9, 1994, Kyoto, Japan.

Nadas, A. 1985. On Turings's Formula for Word Probabilities. IEEE Transactions on Acoustics, Speech and Signal Processing, Vol. ASSP-33, paragraph. 1414-1416, December, 1985.

Ohno, S. and M. Hamanish. 1981. Kadokawa Ruigo Shin-jiten (Kadokawa New Word Category Dictionary). Kadokawa Shoten. January 30, 1981.

Richardson, S., L. Vanderwende, and W. Dolan. 1993. Combining Dictionary-based and Example-based Methods for Natural Language Analysis. In *Proceedings of the Fifth International Conference on Theoretical and Methodological Issues in Machine Translation.* Kyoto, Japan, July 14-16, 1993.

Schuetze, H. 1993. Word space. In *Advances in Neural Information Processing Systems 5*, ed. by Stephen J. Hanson et al, San Mateo, CA: Morgan Kaufmann.

Walker, M., M. Iida, and S. Cote. 1992. Japanese Discourse and the Process of Centering. University of Pennsylvania Tecnical Report, IRCS Report 92-14.

APPENDIX 1: COOC.LISP Program Code

APPENDIX 2: KEE.LISP Program Code

LINE # | TEXT

```
(in-package 'USER)
;;;NOTE!!!Runs in Lucid Common Lisp 4.1, not 4.0.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;===================================================================================
;;;          CO-OC: Building Knowledge Bases of Co-occurrence Information
;;;===================================================================================
;;;
;;;                                 Mark Seligman
;;;                                  Dec. 1994
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;CONTENTS

;;;I. READING/WRITING, PART-OF-SPEECH TABLE, OTHER AUX
;;;  A. READING/WRITING
;;;  B. PART-OF-SPEECH TABLE
;;;  C. OTHER AUX

;;;II. CO-OC MAIN FNS
;;;  A. HACKS AND TOP LEVEL CONTROL
;;;    1.  HACKS
;;;    2.  TOP LEVEL CONTROL
;;;  B. MORPHRECS
;;;    1.  DEFINING MORPHREC DATASTRUCTURE
;;;    2.  MAKING .MORPHS FILE FROM TRANSCRIPT FILE
;;;    3.  SEPARATING SEGMENTS
;;;    4.  MAKING FIRST DRAFT OF MORPHRECS-INDEXED: SORTING MORPHS BY POS (PART OF SPEECH)
;;;    5.  FINALIZING MORPHRECS-INDEXED, MAKING MORPHRECS-SORTED
;;;    6.  SORTING MORPHRECS-INDEXED AND MORPHRECS-SORTED
;;;    7.  QUERYING MORPHRECS-SORTED
;;;    8.  QUERYING MORPHRECS-INDEXED: INCLUDES FINDING SEGMENT MATES
;;;    9.  DEFINING MORPHREC-DIFFREC
;;;   10.  COMPARING MORPHRECS-SORTED
;;;  C. MORPH-COOCRECS
;;;    1.  DEFINING MORPH-COOCREC
;;;    2.  MAKING MORPH-COOCRECS-INDEXED
;;;    3.  MAKE-MORPH-COOCREC-TAIL
;;;    4.  QUERYING MORPH-COOCRECS-INDEXED
;;;    5.  MORPH-WINDOW-MATE QUERY FUNCTIONS
;;;    6.  MAKING MORPH-COOCRECS-SORTED
;;;    7.  SORTING MORPH-COOCRECS-SORTED
;;;    8.  QUERYING MORPH-COOCRECS-SORTED
;;;    9.  DEFINING MORPH-COOCREC-DIFFREC
;;;   10.  COMPARING MORPH-COOCRECS-SORTED
;;;  D. DEFINING NODE-STRUCTURE
;;;  E. CATRECS
;;;    1.  DEFINING CATREC DATASTRUCTURE
;;;    2.  MAKING CATRECS.SORTED
;;;    3.  QUERYING CATRECS-SORTED
;;;    4.  DEFINING CATREC-DIFFREC
;;;    5.  COMPARING CATRECS-SORTED
;;;    6.  MAKING CATRECS-INDEXED
;;;    7.  SORTING CATRECS-INDEXED
;;;    8.  QUERYING CATRECS-INDEXED: INCLUDES FINDING WINDOW MATES
;;;  F. CAT-COOCRECS
;;;    1.  DEFINING CAT-COOCREC DATASTRUCTURE
;;;    2.  MAKING CAT.COOCRECS.INDEXED
;;;    3.  MAKE-CAT-COOCREC-TAIL
;;;    4.  QUERYING CAT-COOCRECS-INDEXED
;;;    5.  CAT-WINDOW-MATE QUERY FUNCTIONS
;;;    6.  MAKING CAT-COOCRECS-SORTED
;;;    7.  SORTING CAT-COOCRECS-SORTED
;;;    8.  QUERYING CAT-COOCRECS-SORTED
;;;    9.  DEFINING CAT-COOCREC-DIFFREC
;;;   10.  COMPARING CAT-COOCRECS-SORTED
;;;  G. THESAURUS, MORPHS-TO-CATS, CATS-TO-MORPHS
;;;    1.  DEFINING THESAURUS-ENTRY
;;;    2.  THESAURUS LOAD AND ACCESS
;;;    3.  THESAURUS HIERARCHY LOAD AND ACCESS
;;;    4.  MAKING MORPHS-TO-CATS
;;;    5.  QUERYING MORPHS-TO-CATS
;;;    6.  GETTING MORPH DICTIONARY FORMS
;;;    7.  MAKING CATS-TO-MORPHS
;;;    8.  QUERYING CATS-TO-MORPHS

;;;III. KEE CONVERSION

;;;***********************************************************************************
;;;
;;; I. READING/WRITING, PRETTY-PRINTING, OTHER AUX
;;;
;;;***********************************************************************************

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; I.A. READING/WRITING
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;To avoid problems with pprint, etc.
(setq *print-level* 20)
(setq *print-shared* nil)
(setq *print-circle* nil)

;;;Reads one s-exp from file.
;;;If no such file, returns nil.
(defun read-from-file (file)
  (with-open-file
      (input-stream file :direction :input :if-does-not-exist nil)
    (if input-stream
        (read input-stream)
        nil)))

;;;Writes (pprints) one s-exp to file.
(defun write-to-file (output file)
  (with-open-file
      (output-stream
       file
       :direction :output
       :if-exists :new-version
       :if-does-not-exist :create)
    (pprint output output-stream)))

;;;Writes (but does not pprint) one s-exp to file.
(defun write-to-file-no-pp (output file)
  (with-open-file
      (output-stream file
```

LINE # TEXT

```
          :direction :output
          :if-exists :new-version
          :if-does-not-exist :create)
      (format output-stream "~s" output)))

;;;Takes SERIES of s-exps from file and returns LIST of s-exps.
(defun read-loop (file)
  (let (output)
    (with-open-file
        (input-stream file :direction :input)
      (let ((new-value t))
        (loop while new-value do
              (setq new-value (read input-stream nil))
              (setq output (cons new-value output)))
        (reverse (cdr output))))))

;;;Takes LIST of s-exps and writes them into a file as SERIES, without parens.
;;;If non-nil numbers?, prints ;;;No.X before each element.
(defun write-loop (list file &optional numbers?)
  (with-open-file
      (output-stream
       file
       :direction :output
       :if-exists :new-version
       :if-does-not-exist :create)
    (let ((count 1))
      (loop for x in list do
            (if numbers?
                (progn
                  (format output-stream "~2%;;;No.~S~%" count)
                  (setq count (1+ count))))
            (print x output-stream)))))

;;;Special aux fn for co-oc
(defun get-global-value-from-file (file attribute)
  (let* ((all-values (read-loop file))
         (fetched (assoc attribute all-values :test #'equal)))
    (second fetched)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  I.B. PART-OF-SPEECH TABLE
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-code-english-symbol (code-number table)
   (fourth (assoc code-number (eval table))))

(defun get-code-japanese-symbol (code-number table)
   (second (assoc code-number (eval table))))

;;;Table giving labeling codes for parts of speech
;;;Japanese glosses follow table of contents, TR-IT-0009.
;;;English glosses are by Seligman, with advice of staff.
(defvar *pos-table*
  '(( 11 固有名詞 |こゆうめいし| proper-noun "30" "固名詞" "固有名詞" )
    ( 12 サ変名詞 |サへんめいし| sahen-noun "05" "サ名詞" "サ変名詞") ;sa-irregular-noun
    ( 13 形容名詞 |けいようめいし| adjectival-noun "31" "形名詞" "形容名詞")
    ( 14 普通名詞 |ふつうめいし| common-noun "04" "普名詞" "普通名詞")
    ( 15 数詞 |すうし| numeral "07" "数  詞" "数詞")
    ( 16 代名詞 |だいめいし| pronoun "06" "代名詞" "代名詞")
    ( 17 人名 |じんめい| personal-name nil nil nil) ;na
    ( 18 住所名 |じゅうしょめい| address-name nil nil nil) ;na
    ( 19 日時 |にちじ| date-or-time nil nil nil) ;na
    ( 21 本動詞 |ほんどうし| verb "32" "本動詞" "本動詞") ;not aux or supp
    ( 22 補助動詞 |ほじょどうし| supplementary-verb "19" "補動詞" "補助動詞") ;eg with te form, eg itadak
    ( 30 語尾 |ごび|  inflectional-ending nil nil nil) ;na
    ( 40 形容詞 |けいようし| adj "01" "形容詞" "形容詞")
    ( 50 副詞 |ふくし| adverb "08" "副  詞" "副詞")
    ( 60 連体詞 |れんたいし| adnoun "09" "連体詞" "連体詞") ;like det, but never alone; non-conjugated adj
    ( 70 接続詞 |せつぞくし| connective-part "10" "接続詞" "接続詞") ;relates clauses, conjunction
    ( 80 感動詞 |かんどうし| interjection "11" "感動詞" "感動詞") ;e.g. hai or, exclamation
    ( 90 間投詞 |かんとうし| filled-pause "33" "間投詞" "間投詞") ;interjection
    (100 助動詞 |じょどうし| aux-verb "12" "助動詞" "助動詞")
    (111 格助詞 |かくじょし| case-part "15" "格助詞" "格助詞")
    (112 準体助詞 |じゅんたいじょし| quasi-noun-part "34" "準助詞" "準体助詞") ;no n
    (113 係助詞 |かかりじょし| topic-part "36" "係助詞" "係助詞") ;wa
    (114 副助詞 |ふくじょし| adverbial-part "13" "副助詞" "副助詞")
    (115 並立助詞 |へいりつじょし| coord-part "35" "並助詞" "並列助詞") ;to to ka
    (116 接続助詞 |せつぞくじょし| connective-part "14" "接助詞" "接続助詞" nil nil nil) ;relates clauses ????
    (117 終助詞 |しゅうじょし| final-part "16" "終助詞" "終助詞")
    (118 連体助詞 |れんたいじょし| adnominal-part nil nil nil) ;na noun-link-particle ??
    (119 引用助詞 |いんよう| quotation-marker nil nil nil) ;to na
    (120 接頭辞 |せっとうじ| prefix "18" "接頭辞" "接頭辞")
    (130 接尾辞 |せつびじ| suffix "17" "接尾辞" "接尾辞")
    (140 その他 |そのほか| etc "99" "その他" "その他")))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  I.C. OTHER AUX
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;Global setting for output accuracy.
(defvar *significant-digits* 4)

(defun round-to-n-digits (number n)
  (if (> n 12)
      (error "asked for more than 12 significant digits"))
  (let ((num (float number))
        (factor
         (if (eq n 0.0)
             0.0
             (expt 10 n))))
    (if (equal factor 0.0)
        (fround num)
        (/ (fround (* factor num)) factor))))

(defun make-numbered-morpheme-group-file (input-file output-file)
  (let* ((input (read-loop input-file))
         (counter 0)
         output)
    (loop for morpheme-group in input do
          (setq output
                (cons
                 (reverse (cons counter (reverse morpheme-group)))
                 output))
          (setq counter (1+ counter)))
    (write-loop (reverse output) output-file)))

;;;If DESTINATION is t, format to standard output, else format to indicated stream.
(defmacro smart-format (destination control-string &rest arguments)
  `(if (equal ,destination t)
     (format ,destination ,control-string ,@arguments)
```

```
    (with-open-file
        (output-stream ,destination
         :direction :output
         :if-exists :append
         :if-does-not-exist :create)
      (format output-stream ,control-string ,@arguments))))

(defun string-remove-spaces (string)
  (let ((list (coerce string 'list))
        (result nil))
    (loop for x in list do
          (if (not (eq x #\space))
              (setq result (cons x result))))
    (coerce (reverse result) 'string)))

(defun empty-string? (string)
  (string-equal (string-remove-spaces string) ""))

(defun list-contains-repetitions? (list)
  (let ((found? nil))
    (loop for x in list until found? do
          (let ((list-minus-x (remove x list :test #'equal)))
            (if (> (- (length list) (length list-minus-x)) 1)
                (setq found? t))))
    found?))

;;;****************************************************************************
;;;
;;;  II. CO-OC MAIN FNS
;;;
;;;****************************************************************************

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  II.A. HACKS AND TOP LEVEL CONTROL
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;============================================================================
;;;  II.A.1.  HACKS
;;;============================================================================

;;;for demo

;;;(defun GET-MORPH-WINDOW-MATES-WITH-FILES n pos-1 lex-1 pos-2 directory corpus-prefix
;;;        &key (threshold nil)(threshold-key 'conditional-probability)(filter-self t)(db-pos-display 'english)
;;;        (return-morphs-only nil)
;;;(get-morph-window-mates-with-files 2 'common-noun '駅 'common-noun "~/survey/jap/data/" "1a.8b")
;;;(get-morph-window-mates-with-files 2 'common-noun '駅 'verb "~/survey/jap/data/" "1a.8b")
;;;(get-morph-window-mates-with-files 2 'common-noun '駅 'verb "~/survey/jap/data/" "1a.8b" :return-morphs-only t)

;;;(defun GET-MORPH-WINDOW-MATES-VIA-CATS-WITH-FILES
;;;    (n pos-1 lex-1 directory corpus-prefix morphs-to-cats-prefix
;;;        &key
;;;        (level nil)
;;;        (threshold nil)
;;;        (threshold-key 'conditional-probability)
;;;        (filter-self t)
;;;        (thesaurus 'thesaurus-full)
;;;        (return-morphs-only nil)
;;;        (cats nil))
;;;(get-morph-window-mates-via-cats-with-files 2 'common-noun '駅 'common-noun "~/survey/jap/data/" "1a.8b" "1a.8b" :return-morphs-only t)
;;;(get-morph-window-mates-via-cats-with-files 2 'common-noun '駅 'verb "~/survey/jap/data/" "1a.8b" "1a.8b" :return-morphs-only t)

;;;(defun get-morph-window-mates-most-specific-with-files
;;;    (n pos-1 lex-1 pos-2 directory corpus-prefix morphs-to-cats-prefix
;;;        &key (level nil)(threshold nil)(threshold-key 'conditional-probability)(filter-self t)
;;;        (thesaurus 'thesaurus-full)
;;;        (db-pos-display 'english)
;;;        (return-morphs-only nil))
;;;(get-morph-window-mates-most-specific-with-files 2 'common-noun '駅 'common-noun "~/survey/jap/data/" "1a.8b" "1a.8b" :return-morphs-only t
)
;;;(get-morph-window-mates-most-specific-with-files 2 'common-noun '駅 'verb "~/survey/jap/data/" "1a.8b" "1a.8b" :return-morphs-only t)
;;;(get-morph-window-mates-most-specific-with-files 2 'common-noun '車 'verb "~/survey/jap/data/" "1a.8b" "1a.8b" :return-morphs-only t)
;;;(get-morph-window-mates-most-specific-with-files 2 'common-noun '車 'common-noun "~/survey/jap/data/" "1a.8b" "1a.8b" :return-morphs-only t
)






;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;List of transcript files in first Japanese emmi corpus.
     ;;;"1a-AMWX"
     ;;;"1b-AMWX"
     ;;;"2a-ATMX"
     ;;;"2b-ATMX"
     ;;;"3a-ATUX"
     ;;;"3b-ATUX"
     ;;;"4a-AKTX"
     ;;;"4b-AKTX"
     ;;;"5a-AYKX"
     ;;;"5b-AYKX"
     ;;;"6a-AHKX"
     ;;;"6b-AHKX"
     ;;;"7a-AYUX"
     ;;;"7b-AYUX"
     ;;;"8a-ATNX"
     ;;;"8b-ATNX"

(defun whole-and-halves ()
  (whole-corpus)
  (halves))

(defun quarters-and-eighths ()
  (quarters)
  (eighths))

(defun whole-corpus ()
  (handle-corpus
   '(2 2)
   '(common-noun verb)
   '(common-noun verb)
   "~/survey/jap/data/"
   '("1a-AMWX"
     "1b-AMWX"
     "2a-ATMX"
     "2b-ATMX"
     "3a-ATUX"
     "3b-ATUX"
     "4a-AKTX"
```

LINE # TEXT

```
      "4b-AKTX"
      "5a-AYKX"
      "5b-AYKX"
      "6a-AHKX"
      "6b-AHKX"
      "7a-AYUX"
      "7b-AYUX"
      "8a-ATNX"
      "8b-ATNX")
    "1a.8b"
    nil))

(defun halves ()
  (handle-corpora-and-comparisons
   '(2 2)
   '(common-noun verb)
   '(common-noun verb)
   "~/survey/jap/data/"
   '("1a-AMWX"
     "1b-AMWX"
     "2a-ATMX"
     "2b-ATMX"
     "3a-ATUX"
     "3b-ATUX"
     "4a-AKTX"
     "4b-AKTX" )

   '("5a-AYKX"
     "5b-AYKX"
     "6a-AHKX"
     "6b-AHKX"
     "7a-AYUX"
     "7b-AYUX"
     "8a-ATNX"
     "8b-ATNX")

   "1a.4b"
   "5a.8b"
   "1a.8b"))

(defun quarters ()
  (handle-corpora-and-comparisons
   '(2 2)
   '(common-noun verb)
   '(common-noun verb)
   "~/survey/jap/data/"
   '("1a-AMWX"
     "1b-AMWX"
     "2a-ATMX"
     "2b-ATMX"
     "3a-ATUX"
     "3b-ATUX"
     "4a-AKTX"
     "4b-AKTX"
     "5a-AYKX"
     "5b-AYKX"
     "6a-AHKX"
     "6b-AHKX")

   '("7a-AYUX"
     "7b-AYUX"
     "8a-ATNX"
     "8b-ATNX")

   "1a.6b"
   "7a.8b"
   "1a.8b")
  :make-morphs-files nil)

(defun eighths ()
  (handle-corpora-and-comparisons
   '(2 2)
   '(common-noun verb)
   '(common-noun verb)
   "~/survey/jap/data/"
   '("1a-AMWX"
     "1b-AMWX"
     "2a-ATMX"
     "2b-ATMX"
     "3a-ATUX"
     "3b-ATUX"
     "4a-AKTX"
     "4b-AKTX"
     "5a-AYKX"
     "5b-AYKX"
     "6a-AHKX"
     "6b-AHKX"
     "7a-AYUX"
     "7b-AYUX")

   '("8a-ATNX"
     "8b-ATNX")

   "1a.7b"
   "8a.8b"
   "1a.8b")
  :make-morphs-files nil)

(defun sixteenths ()
  (handle-corpora-and-comparisons
   '(2 2)
   '(common-noun verb)
   '(common-noun verb)
   "~/survey/jap/data/"
   '("1a-AMWX"
     "1b-AMWX"
     "2a-ATMX"
     "2b-ATMX"
     "3a-ATUX"
     "3b-ATUX"
     "4a-AKTX"
     "4b-AKTX"
     "5a-AYKX"
     "5b-AYKX"
     "6a-AHKX"
     "6b-AHKX"
     "7a-AYUX"
     "7b-AYUX"
     "8a-ATNX")
```

```
    '("8b-ATNX")

    "1a.8a"
    "8b"
    "1a.8b")
   :make-morphs-files nil)

;;;For quick benchmark test of whole system.
(defun 1a.vs.1b ()
  (handle-corpora-and-comparisons
    '(2 2)
    '(common-noun verb)
    '(common-noun verb)
    "~/survey/jap/data/"
    '("1a-AMWX")
    '("1b-AMWX")

    "1a"
    "1b"
    nil))

;;;============================================================================
;;;  II.A.1.  TOP LEVEL CONTROL
;;;============================================================================

;;;Handles two corpora and compares them.
(defun handle-corpora-and-comparisons
    (n-range
     pos-1-list
     pos-2-list
     directory                          ;like "~/survey/jap/data/"
     transcript-file-prefixes-1         ;like ("1a-AMWX" "1b-AMWX")
     transcript-file-prefixes-2         ;like ("2a-AMWX" "2b-AMWX")
     corpus-prefix-1                    ;like "1a.1b"
     corpus-prefix-2                    ;like "2a.2b"
     morphs-to-cats-prefix              ;if using pre-made files

     &key
     ;;highest-level-control
     (handle-corpus-1 t)
     (handle-corpus-2 t)
     (handle-morph-and-cat-comparisons t)

     ;;for handle-morph-and-cat-comparisons, if permitted
     (handle-morph-comparisons t)
     (handle-cat-comparisons t)

     ;;high-level control
     (handle-morphs t)
     (handle-cats t)

     ;;for handle-morphs, if permitted
     (make-morphs-files t)
     (segment-corpus t)
     (make-morphrecs-indexed-and-sorted t)
     (make-morph-coocrecs-indexed t)
     (make-morph-coocrecs-sorted t)

     (resort-by 'hit-count)
     (separators '(END-OF-TURN))
     (db-pos-display 'english)
     (threshold 0.1)
     (threshold-key 'conditional-probability)

     ;;for handle-cats, if permitted
     (make-catrecs-sorted t)
     (make-catrecs-indexed t)
     (make-cat-coocrecs-sorted t)
     (make-cat-coocrecs-indexed t)

     ;;for handle-mtc-and-ctm if permitted by used of
     ;;morphs-to-cats-prefix
     (thesaurus 'thesaurus-full))

  (if handle-corpus-1
      (handle-corpus
       n-range
       pos-1-list
       pos-2-list
       directory
       transcript-file-prefixes-1
       corpus-prefix-1
       morphs-to-cats-prefix
       :thesaurus thesaurus
       :handle-morphs handle-morphs
       :handle-cats handle-cats
       :make-morphs-files make-morphs-files
       :segment-corpus segment-corpus
       :make-morphrecs-indexed-and-sorted make-morphrecs-indexed-and-sorted
       :make-morph-coocrecs-indexed make-morph-coocrecs-indexed
       :make-morph-coocrecs-sorted make-morph-coocrecs-sorted
       :resort-by resort-by
       :separators separators
       :db-pos-display db-pos-display
       :threshold threshold
       :threshold-key threshold-key
       :make-catrecs-sorted make-catrecs-sorted
       :make-catrecs-indexed make-catrecs-indexed
       :make-cat-coocrecs-sorted make-cat-coocrecs-sorted
       :make-cat-coocrecs-indexed make-cat-coocrecs-indexed))

  (if handle-corpus-2
      (handle-corpus
       n-range
       pos-1-list
       pos-2-list
       directory
       transcript-file-prefixes-2
       corpus-prefix-2
       morphs-to-cats-prefix
       :thesaurus thesaurus
       :handle-morphs handle-morphs
       :handle-cats handle-cats
       :make-morphs-files make-morphs-files
       :segment-corpus segment-corpus
       :make-morphrecs-indexed-and-sorted make-morphrecs-indexed-and-sorted
       :make-morph-coocrecs-indexed make-morph-coocrecs-indexed
       :make-morph-coocrecs-sorted make-morph-coocrecs-sorted
       :resort-by resort-by
       :separators separators
```

LINE # TEXT

```
599         :db-pos-display db-pos-display
600         :threshold threshold
601         :threshold-key threshold-key
602         :make-catrecs-sorted make-catrecs-sorted
603         :make-catrecs-indexed make-catrecs-indexed
604         :make-cat-coocrecs-sorted make-cat-coocrecs-sorted
605         :make-cat-coocrecs-indexed make-cat-coocrecs-indexed))
606
607     (if handle-morph-and-cat-comparisons
608         (handle-morph-and-cat-comparisons
609          directory corpus-prefix-1 corpus-prefix-2
610          :handle-morph-comparisons handle-morph-comparisons
611          :handle-cat-comparisons handle-cat-comparisons)))
612
613 ;;;Handles a single corpus.
614 (defun handle-corpus
615        (n-range
616         pos-1-list
617         pos-2-list
618         directory                          ;like "~/survey/jap/data/"
619         transcript-file-prefixes           ;like ("1a-AMWX" "1b-AMWX")
620         corpus-prefix                      ;like "1a.1b"
621         morphs-to-cats-prefix              ;if using pre-made files
622
623         &key
624         ;;high-level control
625         (handle-morphs t)
626         (handle-cats t)
627
628         ;;for handle-morphs, if permitted
629         (make-morphs-files t)
630         (segment-corpus t)
631         (make-morphrecs-indexed-and-sorted t)
632         (make-morph-coocrecs-indexed t)
633         (make-morph-coocrecs-sorted t)
634
635         (resort-by 'hit-count)
636         (separators '(END-OF-TURN))
637         (db-pos-display 'english)
638         (threshold 0.1)
639         (threshold-key 'conditional-probability)
640
641         ;;for handle-cats, if permitted
642         (make-catrecs-sorted t)
643         (make-catrecs-indexed t)
644         (make-cat-coocrecs-sorted t)
645         (make-cat-coocrecs-indexed t)
646
647         ;;for handle-mtc-and-ctm if permitted by used of
648         ;;morphs-to-cats-prefix
649         (thesaurus 'thesaurus-full))
650
651     (if handle-morphs
652         (handle-morphs
653          n-range
654          pos-1-list
655          pos-2-list
656          directory
657          transcript-file-prefixes
658          corpus-prefix
659          :make-morphs-files make-morphs-files
660          :segment-corpus segment-corpus
661          :make-morphrecs-indexed-and-sorted make-morphrecs-indexed-and-sorted
662          :make-morph-coocrecs-indexed make-morph-coocrecs-indexed
663          :make-morph-coocrecs-sorted make-morph-coocrecs-sorted
664          :resort-by resort-by
665          :separators separators
666          :db-pos-display db-pos-display
667          :threshold threshold
668          :threshold-key threshold-key))
669
670     ;;If MORPHS-TO-CATS prefix is provided, a pre-made MORPHS-TO-CATS file is to be used.
671     ;;In this case, do not make new mapping between morphs and cats;
672     ;;instead, use pointer (MORPHS-TO-CATS-PREFIX) to pre-made one.
673     ;;Else do make new mappings, using CORPUS-PREFIX to name it.
674     (if (null morphs-to-cats-prefix)
675         (let ((pos-list (union pos-1-list pos-2-list)))
676           (format t "~%Handling morphs-to-cats-and-cats-to-morphs.")
677           (handle-morphs-to-cats-and-cats-to-morphs
678            pos-list directory corpus-prefix :thesaurus thesaurus)
679           (setq morphs-to-cats-prefix corpus-prefix)))
680
681     (if handle-cats
682         (handle-cats
683          n-range
684          pos-1-list
685          pos-2-list
686          directory                         ;like "~/survey/jap/data/"
687          corpus-prefix                     ;like "1a.1b"
688          morphs-to-cats-prefix             ;<<
689          :make-catrecs-sorted make-catrecs-sorted
690          :make-catrecs-indexed make-catrecs-indexed
691          :make-cat-coocrecs-sorted make-cat-coocrecs-sorted
692          :make-cat-coocrecs-indexed make-cat-coocrecs-indexed
693          :threshold threshold
694          :threshold-key threshold-key
695          :thesaurus thesaurus)))
696
697 (defun handle-morph-and-cat-comparisons
698        (directory
699         corpus-prefix-1
700         corpus-prefix-2
701         &key
702         (handle-morph-comparisons t)
703         (handle-cat-comparisons t))
704     (if handle-morph-comparisons
705         (progn
706           (format t "~%Handling morph comparisons.")
707           (handle-morph-comparisons directory corpus-prefix-1 corpus-prefix-2)))
708     (if handle-cat-comparisons
709         (progn
710           (format t "~%Handling cat comparisons.")
711           (handle-cat-comparisons directory corpus-prefix-1 corpus-prefix-2))))
712
713 ;;;Handles morphs within one corpus.
714 (defun handle-morphs
715        (n-range
716         pos-1-list
717         pos-2-list
718         directory                          ;like "~/survey/jap/data/"
```

LINE # TEXT

```
719       transcript-file-prefixes          ;like ("1a-AMWX" "1b-AMWX")
720       corpus-prefix                     ;like "1a.1b"
721
722       &key
723       (make-morphs-files t)
724       (segment-corpus t)
725       (make-morphrecs-indexed-and-sorted t)
726       (make-morph-cooccrecs-indexed t)
727       (make-morph-cooccrecs-sorted t)
728
729       (resort-by 'hit-count)
730       (separators '(END-OF-TURN))
731       (db-pos-display 'english)
732       (threshold 0.1)
733       (threshold-key 'conditional-probability))
734
735   ;;MAKE .MORPHS FILES
736   (if make-morphs-files
737       (make-morphs-files-from-transcript-files directory transcript-file-prefixes))
738
739   ;;SEGMENT CORPUS
740   (if segment-corpus
741       (segment-transcript-files-with-files
742        directory corpus-prefix transcript-file-prefixes :separators separators))
743
744   ;;MAKE MORPHRECS-INDEXED-AND-SORTED
745   (if make-morphrecs-indexed-and-sorted
746       (make-morphrecs-indexed-and-sorted-with-files
747        directory corpus-prefix :db-pos-display db-pos-display :resort-by resort-by))
748
749   ;;MAKE MORPH-COOCRECS-INDEXED
750   (if  make-morph-cooccrecs-indexed
751        (make-morph-cooccrecs-indexed-with-files
752         n-range pos-1-list pos-2-list directory corpus-prefix
753         :db-pos-display db-pos-display :threshold threshold :threshold-key threshold-key))
754
755   ;;MAKE MORPH-COOCRECS-SORTED
756   (if make-morph-cooccrecs-sorted
757       (make-morph-cooccrecs-sorted-with-files
758        n-range pos-1-list pos-2-list directory corpus-prefix
759        :db-pos-display db-pos-display :threshold threshold :threshold-key threshold-key)))
760
761 (defun handle-morph-comparisons (directory corpus-prefix-1 corpus-prefix-2)
762        (compare-morphrecs-sorted directory corpus-prefix-1 corpus-prefix-2)
763        (compare-morph-cooccrecs-sorted directory corpus-prefix-1 corpus-prefix-2))
764
765 (defun handle-morphs-to-cats-and-cats-to-morphs
766     (pos-list directory corpus-prefix &key (thesaurus 'thesaurus-full))
767   (make-morphs-to-cats-with-files
768    pos-list directory corpus-prefix :thesaurus thesaurus)
769   (make-cats-to-morphs-with-files directory corpus-prefix :thesaurus thesaurus))
770
771 ;;;Handle cats for one corpus.
772 (defun handle-cats
773     (n-range
774      pos-1-list
775      pos-2-list
776      directory                         ;like "~/survey/jap/data/"
777      corpus-prefix                     ;like "1a.1b"
778      morphs-to-cats-prefix             ;<<
779
780      &key
781      (make-catrecs-sorted t)
782      (make-catrecs-indexed t)
783      (make-cat-cooccrecs-sorted t)
784      (make-cat-cooccrecs-indexed t)
785
786      (threshold 0.1)
787      (threshold-key 'conditional-probability)
788      (thesaurus 'thesaurus-full))
789
790   ;;MAKE CATRECS-SORTED
791   (if make-catrecs-sorted
792       (progn
793         (format t "~%Making catrecs-sorted.~%")
794         (make-catrecs-sorted-with-files
795          directory corpus-prefix morphs-to-cats-prefix
796          :thesaurus thesaurus)))        ;<<
797
798   ;;MAKE CATRECS-INDEXED
799   (if make-catrecs-indexed
800       (progn
801         (format t "~%Making catrecs-indexed.~%")
802         (make-catrecs-indexed-with-files
803          directory corpus-prefix)))
804
805   ;;MAKE CAT-COOCRECS-INDEXED
806   (if make-cat-cooccrecs-indexed
807       (progn
808         (format t "~%Making cat-cooccrecs-indexed.~%")
809         (make-cat-cooccrecs-indexed-with-files
810          n-range pos-1-list pos-2-list directory corpus-prefix morphs-to-cats-prefix ;<<
811          :threshold threshold
812          :threshold-key threshold-key
813          :thesaurus thesaurus)))
814
815   ;;MAKE CAT-COOCRECS-SORTED
816   (if make-cat-cooccrecs-sorted
817       (progn
818         (format t "~%Making cat-cooccrecs-sorted.~%")
819         (make-cat-cooccrecs-sorted-with-files
820          directory corpus-prefix
821          :threshold threshold
822          :threshold-key threshold-key))))
823
824 (defun handle-cat-comparisons (directory corpus-prefix-1 corpus-prefix-2)
825        (compare-catrecs-sorted directory corpus-prefix-1 corpus-prefix-2)
826        (compare-cat-cooccrecs-sorted directory corpus-prefix-1 corpus-prefix-2))
827
828 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
829 ;;;  II.B. MORPHRECS
830 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
831
832 ;;;============================================================================
833 ;;;  II.B.1.  DEFINING MORPHREC DATASTRUCTURE
834 ;;;============================================================================
835
836 (defvar *morphrec-elements*
837   '((lex 0)
838     (pos 1)
```

LINE # | TEXT

```
    (segments 2)
    (hit-count 3)
    (morph-segments-count 4)
    (occur-in-segment-probability 5)
    (occur-in-segment-information 6)
    (occur-in-segment-probability-smoothed 7)
    (morph-unigram-probability 8)
    (morph-unigram-information 9)
    (morph-unigram-probability-smoothed 10)
    (thesaurus-entries 11)))            ;<<<

(defun make-morphrec (lex pos)
  (let ((morphrec
         (loop for element in *morphrec-elements* collect nil)))
    (setq morphrec (put-morphrec-element morphrec 'lex lex))
    (setq morphrec (put-morphrec-element morphrec 'pos pos))
    morphrec))

;;;<<First position is zero, not 1.
(defun get-morphrec-element-position (element-name)
  (second
   (assoc element-name *morphrec-elements*)))

(defun get-morphrec-element (morphrec element-name)
  (let ((position (get-morphrec-element-position element-name)))
    (nth position morphrec)))

(defun put-morphrec-element (morphrec element-name value)
  ;;;Using copy-list, not the more general copy-tree
  (let ((morphrec-copy (copy-list morphrec))
        (position (get-morphrec-element-position element-name)))
    (setf (nth position morphrec-copy) value)
    morphrec-copy))

(defun add-morphrec-element (morphrec element-name value)
  (let ((morphrec-copy (copy-list morphrec))
        (old-value (get-morphrec-element morphrec element-name))
        (new-value nil)
        (position (get-morphrec-element-position element-name)))
    (if (not (listp old-value))
        (error "old value not a list")
        (progn (setq new-value (adjoin value old-value :test #'equal))
               (setf (nth position morphrec-copy) new-value)))
    morphrec-copy))

;;;============================================================================
;;;  II.B.2.  MAKING .MORPHS FILE FROM TRANSCRIPT FILE
;;;============================================================================

;;;Takes EMMI transcript file, converts to .morphs file,
;;;similar to atr dialogue database format.
;;;Adds end-of-turn lines.
;;;Can handle output of Kitagawa's origami program,
;;;which inserts pause information.
;;;(はい 慣用句)
;;;(人工知能学会国際会議事務局 感動詞)
;;;(で NIL)
;;;(す 固有名詞)
;;;(END-OF-TURN)
;;;(あ NIL)
;;;(PAUSE 0.3221 nil)
;;;(あのー NIL)
;;;(PAUSE 0.5399 NIL)
;;;(えっと NIL)
;;;Errors are included as
;;;((こう) NIL)
;;;but aizuichi in curly brackets are omitted.
;;;Preprocess hesitations like *AIZUCHI*, then expand into multiple lines
;;;during post-processing.
;;;Ignores empty lines in transcript file
;;;TRANSCRIPT-FILES is list of filenames without suffixes.
(defun make-morphs-files-from-transcript-files (directory transcript-file-prefixes)
  (loop for transcript-file-prefix in transcript-file-prefixes do
        (let ((transcript-file (format nil "~A~A" directory transcript-file-prefix))
              (morphs-file (format nil "~A~A.morphs" directory transcript-file-prefix)))

          ;;Make .morphs file from transcript file
          (format t "~%converting to db format: ~s" morphs-file)
          (make-morphs-file-from-transcript transcript-file morphs-file))))

(defun make-morphs-file-from-transcript
    (input-file output-file &key
                (db-pos-display 'english)
                (filter-aizuchi? 't))

  (with-open-file (output-stream
                   output-file
                   :direction :output
                   :if-exists :new-version
                   :if-does-not-exist :create)
    (with-open-file (input-stream input-file :direction :input)

      (do ((line (read-line input-stream nil 'eof)
                 (read-line input-stream nil 'eof)))
          ((eq line 'eof) t)

        ;;If line gave empty string (null or only spaces) do nothing --
        ;;go around again and read another line.
        (if (not (empty-string? line))
            ;;Compute FILTERED-MORPHS-WITH-CODES.
            ;;In order to match codes one-to-one, we filter
            ;;pauses and errors. Note that aizuchi and hesitations remain,
            ;;since they have part of speech codes.
            (let* ((morphs
                    ;;Calls FILTER-LINE-STRING, which turns most punctuation into space.
                    ;;But round, curley and square brackets (errors, aizuchi, hesitations) become round brackets,
                    ;;ie lists. And aizuchi and hesitation lists are marked using *aizuchi* or *hesitation*
                    ;;as first element.
                    (break-line-into-symbols line))
                   ;;<<removes numbers
                   (morphs-minus-pauses (remove-numbers-from-list morphs))
                   (filtered-morphs
                    ;;<<<removes lists other than aizuchi or hesitations
                    (remove-errors-from-list morphs-minus-pauses))
                   (codes nil)
                   (filtered-morphs-with-codes nil))

              ;;Check for missing transcript line.
              (let ((first-element (first filtered-morphs)))
                (if (null first-element)
```

LINE # TEXT

```
                    (progn
                      (format t "~%line: ~s" line)
                      (format t "~%filtered morphs: ~s" filtered-morphs)
                      (error "Missing transcript line?  Filtered-morphs contained only numbers."))))

              ;;Read lines until code line is found, ignoring empty lines.
              (loop until codes do
                    (let ((line (read-line input-stream nil 'eof)))
                      (cond ((and
                              (stringp line)
                              (not (empty-string? line)))
                             (setq codes (break-line-into-symbols line)))
                            ((eq line 'eof)
                             (error "Missing code line at end of file.")))))

              ;;Check for missing or bad code line.
              (if (not (numberp (first codes)))
                  (progn
                    (format t "~%line: ~s" line)
                    (format t "~%codes: ~s" codes)
                    (error "Missing codes line? First element of line is not a number.")))

              (let ((filtered-morphs-length (length filtered-morphs))
                    (codes-length (length codes)))
                (if (not (equal filtered-morphs-length codes-length))
                    (progn (format t "~%filtered-morphs: ~s~%length: ~s~%codes: ~s~%length: ~s~%"
                                   filtered-morphs filtered-morphs-length codes codes-length)
                           (error "Not equal lengths."))))

              ;;Replace code numbers with English or Japanese symbols if required.
              (loop for filtered-morph in filtered-morphs do
                    (case db-pos-display
                      ((japanese)
                       (setq filtered-morphs-with-codes
                             (cons (list filtered-morph
                                         (get-code-japanese-symbol (first codes) '*pos-table*))
                                   filtered-morphs-with-codes))
                       (setq codes (rest codes)))
                      ((english)
                       (setq filtered-morphs-with-codes
                             (cons (list filtered-morph
                                         (get-code-english-symbol (first codes) '*pos-table*))
                                   filtered-morphs-with-codes))
                       (setq codes (rest codes)))

                      ((number)
                       (setq filtered-morphs-with-codes
                             (cons (list filtered-morph
                                         (first codes))
                                   filtered-morphs-with-codes))
                       (setq codes (rest codes)))
                      (t (error "bad db-pos-display value"))))

              (setq filtered-morphs-with-codes (reverse filtered-morphs-with-codes))

              ;;Now loop thru original, NON-FILTERED morphs.
              (loop for morph in morphs do
                    (cond
                      ;;Pauses (numbers) and errors (lists without ** identifiers) go to output
                      ;;as (XX NIL), with NIL showing that there is no code.
                      ((or (numberp morph)
                           (is-error? morph))
                       (format output-stream "(~s ~s)~%" morph nil))

                      ;;But real morphs, aizuchi, and hesitations do have codes,
                      ;;so we print from (and maintain) FILTERED-MORPHS-WITH-CODES
                      ;;instead of from original morphs.
                      (t
                       (format output-stream "~s~%" (first filtered-morphs-with-codes))
                       (setq filtered-morphs-with-codes (rest filtered-morphs-with-codes)))))

              ;;This is the end of one transcript line representing one turn, so
              ;;add END-OF-TURN, except if line was empty due to error in input file.
              (if morphs (format output-stream "(end-of-turn)~%")))))

  ;;Post-processing.
  ;;Add "pause" before numbers
  ;;Omit symbols *AIZUCHI* and *HESITATION*
  ;;Filter AIZUCHI lines unless explicitly preserved.
  ;;Expand HESITATIONS and preserved AIZUCHI, giving one output line for each separate symbol.
  (let ((morphs (read-loop output-file))
        output)
    (if (probe-file output-file)
        (delete-file output-file))
    (loop for morph in morphs do
          (cond ((numberp (first morph))
                 (setq morph (cons 'pause morph))
                 (setq output (cons morph output)))
                ((and (consp (first morph))
                      (equal (first (first morph)) '*AIZUCHI*))
                 ;;By default, FILTER AIZUCHI at this stage.
                 (if filter-aizuchi?
                     nil
                     (let ((expansions (expand-hesitations-or-aizuchi morph)))
                       (loop for expansion in expansions do
                             (setq output (cons expansion output))))))
                ((and (consp (first morph))
                      (equal (first (first morph)) '*HESITATION*))
                 (let ((expansions (expand-hesitations-or-aizuchi morph)))
                   (loop for expansion in expansions do
                         (setq output (cons expansion output)))))
                (t (setq output (cons morph output)))))
    (write-loop (reverse output) output-file)))

;;;Recieves entire morph, e.g. ((*HESITATION* ettou anou) 90)
;;;Eliminates id symbol, then makes one line for each item, e.g.
;;;((ettou 90)(anou 90))
(defun expand-hesitations-or-aizuchi (morph)
  (let ((hesitations-or-aizuchi (rest (first morph)))
        (code-or-symbol (first (rest morph))))
    (loop for item in hesitations-or-aizuchi collect
          (list item code-or-symbol))))

(defun is-error? (element)
  (and (consp element)
       (and (not (equal (first element) '*aizuchi*))
            (not (equal (first element) '*hesitation*)))))

(defun break-line-into-symbols (line-string)
  (if (stringp line-string)
```

```
LINE #  TEXT
1079        (let* ((filtered-line-string (filter-line-string line-string))
1080               (filtered-line-list (string-to-list filtered-line-string)))
1081          filtered-line-list)
1082        line-string))
1083
1084  (defun remove-numbers-from-list (list)
1085    (let ((output nil))
1086      (loop for x in list do
1087            (if (not (numberp x))
1088                (setq output (cons x output))))
1089      (reverse output)))
1090
1091  (defun remove-errors-from-list (list)
1092    (let ((output nil))
1093      (loop for x in list do
1094            (if (not (is-error? x))
1095                (setq output (cons x output))))
1096      (reverse output)))
1097
1098  (defun string-to-list (string)
1099    (let (output)
1100      (with-input-from-string (my-stream string)
1101        (do ((current (read my-stream nil 'eof)
1102                      (read my-stream nil 'eof)))
1103            ((eq current 'eof) t)
1104          (setq output (cons current output))))
1105      (reverse output)))
1106
1107  (defun filter-line-string (string)
1108    (if (stringp string)
1109        (let* (output
1110               (beheaded-string (behead-string string #\: )) ;<<<<<
1111               (list (coerce beheaded-string 'list)))
1112          (loop for e in list do
1113                (cond ((or (alpha-char-p e)
1114                           (eq #\: e)     ;romaji punct
1115                           (eq #\* e)
1116                           (eq #\/ e)
1117
1118                           (eq #\   e)    ;<<<kanji space
1119                           (eq #\:  e)
1120                           (eq #\／ e))
1121                       (setq output (cons #\Space output)))
1122                      ((eq #\（ e)
1123                       (setq output (cons #\( output)))
1124                      ;;aizuchi [ hai ] appears as (*aizuchi* hai)
1125                      ((eq #\［ e)
1126                       (setq output (cons #\( output))
1127                       (setq output (append (reverse (coerce "*AIZUCHI*" 'list)) output)))
1128                      ;;hesitation [anou ettou] appears as (*hesitation* anou ettou)
1129                      ((eq #\[ e)
1130                       (setq output (cons #\( output))
1131                       (setq output (append (reverse (coerce "*HESITATION*" 'list)) output))) ;<<<
1132                      ((or (eq #\)  e)
1133                           (eq #\］ e)
1134                           (eq #\] e))   ;<<
1135                       (setq output (cons #\) output)))
1136                      (t (setq output (cons e output)))))
1137          (coerce (reverse output) 'string))
1138        string))
1139
1140  (defun behead-string (string end-of-head-char)
1141    (if (stringp string)
1142        (let ((position (position end-of-head-char string :test #'char-equal)))
1143          (if position
1144              (subseq string
1145                      (1+ position)
1146                      (length string))
1147              string))
1148        string))
1149
1150  ;;;============================================================================
1151  ;;;  II.B.3.  SEPARATING SEGMENTS
1152  ;;;============================================================================
1153
1154  (defun segment-transcript-files-with-files
1155      (directory corpus-prefix transcript-file-prefixes &key (separators '(END-OF-TURN)) (make-morphs-files nil))
1156    (let* ((segments-file (format nil "~A~A.segments" directory corpus-prefix))
1157           (segment-transcript-files-results
1158            (segment-transcript-files directory transcript-file-prefixes :separators separators :make-morphs-files make-morphs-files))
1159           (all-segments (first segment-transcript-files-results))
1160           (discourse-final-segments (second segment-transcript-files-results)))
1161      (write-to-file-no-pp (list all-segments discourse-final-segments) segments-file)))
1162
1163  (defun segment-transcript-files
1164      (directory transcript-file-prefixes &key (separators '(END-OF-TURN)) (make-morphs-files nil))
1165    (if make-morphs-files
1166        (make-morphs-files-from-transcript-files directory transcript-file-prefixes))
1167
1168    (let ((all-segments nil)
1169          (discourse-final-segments nil))
1170      ;;Assume morphs files are complete.
1171      (loop for transcript-file-prefix in transcript-file-prefixes do
1172            (let ((morphs-file (format nil "~A~A.morphs" directory transcript-file-prefix)))
1173
1174              ;;Separate .morphs file into segments
1175              (format t "~%separating segments: ~s" morphs-file)
1176              (let ((segments
1177                     (separate-segments morphs-file
1178                                        :separators separators)))
1179                (setq all-segments (append all-segments segments))
1180                (setq discourse-final-segments
1181                      ;;first segment is 1, not 0
1182                      (cons (length all-segments)
1183                            discourse-final-segments)))))
1184      (format t "~%SEGMENT-TRANSCRIPT-FILES completed. ~%NUMBER OF SEGMENTS: ~s ~%DISCOURSE-FINAL-SEGMENTS: ~s"
1185              (length all-segments) discourse-final-segments)
1186      (list all-segments discourse-final-segments)))
1187
1188  (defun separate-segments (input-file &key (separators '(|.| |?|)))
1189    (let* ((input (read-loop input-file))
1190           this-segment
1191           segments)
1192      (loop for morpheme in input do
1193            (let ((morph (first morpheme)))
1194              (setq this-segment (cons morpheme this-segment))
1195              (if (member morph separators)
1196                  (progn (setq segments
1197                               (cons (reverse this-segment) segments))
1198                         (setq this-segment nil)))))
```

```
1199      (reverse segments)))
1200
1201    ;;;============================================================================
1202    ;;;  II.B.4.  MAKING FIRST DRAFT OF MORPHRECS-INDEXED: SORTING MORPHS BY POS (PART OF SPEECH)
1203    ;;;============================================================================
1204
1205    (defun make-morphrecs-indexed-and-sorted-with-files
1206        (directory corpus-prefix &key (db-pos-display 'english) (resort-by 'hit-count))
1207
1208      ;;Sort corpus by pos, where each POS-RECORD shows segments for each lex.
1209      ;;PRELIMINARY MORPHRECS-INDEXED is made during this sorting process.
1210      ;;It is then AUGMENTED and RESORTED.
1211      (let* ((segments-file (format nil "~A~A.segments" directory
1212                                    corpus-prefix))
1213             (segments-info (read-from-file segments-file))
1214             (all-segments (first segments-info))
1215             (discourse-final-segments (second segments-info))
1216             (sort-results
1217              (sort-corpus-by-pos all-segments
1218                                  :db-pos-display db-pos-display
1219                                  :return-extra-info? t))
1220             (all-segments-count (first sort-results))
1221             (corpus-length-in-morphemes (second sort-results))
1222             (number-of-different-morphemes (third sort-results))
1223
1224             ;;PRELIMINARY MORPHRECS-INDEXED
1225             (morphrecs-indexed (fourth sort-results))
1226
1227             ;;AUGMENT MORPHRECS-INDEXED
1228             (augment-morphrecs-indexed-results
1229              (augment-morphrecs-indexed
1230               morphrecs-indexed
1231               all-segments-count
1232               corpus-length-in-morphemes
1233               number-of-different-morphemes))
1234             (augmented-morphrecs-indexed (first augment-morphrecs-indexed-results))
1235
1236             ;;RESORT MORPHRECS-INDEXED
1237             (resorted-augmented-morphrecs-indexed
1238              (resort-morphrecs-indexed augmented-morphrecs-indexed :resort-by resort-by))
1239
1240             ;;RETRIEVE MORPHRECS-SORTED
1241             (morphrecs-sorted (second augment-morphrecs-indexed-results))
1242             (resorted-morphrecs-sorted (sort-morphrecs morphrecs-sorted resort-by))
1243
1244             ;;get morph-unigram-entropy (presently using raw, not smoothed probabilities)
1245             (morph-unigram-entropy
1246              (round-to-n-digits
1247               (get-morphrec-element-entropy 'morph-unigram-probability 'morph-unigram-information
1248                                             morphrecs-sorted)
1249               *significant-digits*)))
1250
1251        (format t "~%ALL-SEGMENTS-COUNT: ~s~%CORPUS-LENGTH-IN-MORPHEMES: ~s~%NUMBER-OF-DIFFERENT-MORPHEMES: ~s~%"
1252                all-segments-count corpus-length-in-morphemes number-of-different-morphemes)
1253
1254        (let ((morphs-global-file (format nil "~A~A.morphs.global" directory corpus-prefix))
1255              (morphrecs-indexed-file (format nil "~A~A.morphrecs.indexed" directory corpus-prefix))
1256              (morphrecs-sorted-file (format nil "~A~A.morphrecs.sorted" directory corpus-prefix)))
1257
1258          ;;Save MORPHRECS-GLOBAL
1259          (if (probe-file morphs-global-file)
1260              (delete-file morphs-global-file))
1261
1262          (smart-format morphs-global-file "~%~s" (list 'all-segments-count all-segments-count))
1263          (smart-format morphs-global-file "~%~s" (list 'discourse-final-segments discourse-final-segments))
1264          (smart-format morphs-global-file "~%~s" (list 'corpus-length-in-morphemes corpus-length-in-morphemes))
1265          (smart-format morphs-global-file "~%~s" (list 'number-of-different-morphemes number-of-different-morphemes))
1266          (smart-format morphs-global-file "~%~s" (list 'morph-unigram-entropy morph-unigram-entropy))
1267
1268          ;;Save MORPHRECS-INDEXED
1269          (format t "Writing to ~S~%" morphrecs-indexed-file)
1270          (write-to-file-no-pp resorted-augmented-morphrecs-indexed morphrecs-indexed-file)
1271
1272          ;;Save MORPHRECS-SORTED
1273          (format t "Writing to ~S~%" morphrecs-sorted-file)
1274          (write-to-file-no-pp resorted-morphrecs-sorted morphrecs-sorted-file))))
1275
1276    (defun sort-corpus-by-pos (list-of-segments &key (db-pos-display 'english) (return-extra-info? nil))
1277      (format t "~%sorting entire corpus by pos")
1278      (let ((morphrecs-indexed
1279             (loop for p in *pos-table* collect
1280                   (list
1281                    (case db-pos-display
1282                      ((number) (first p))
1283                      ((english)(fourth p))
1284                      ((japanese (second p))))
1285                    nil))))
1286
1287        ;;For every segment, for every morpheme, put it in the right
1288        ;;sublist, with record of the segment it was found in.
1289        (let ((all-segments-count 0)
1290              (corpus-length-in-morphemes 0)
1291              (number-of-different-morphemes 0))
1292          (loop for segment in list-of-segments do
1293                (format t ".")
1294                (setq all-segments-count (1+ all-segments-count))
1295                (loop for element in segment do
1296                      (setq corpus-length-in-morphemes (1+ corpus-length-in-morphemes))
1297                      (let* ((element-lex (first element))
1298                             (element-pos (second element))
1299                             (new-pos-record nil))
1300                        (if element-pos
1301                            (let* ((old-pos-record
1302                                    (assoc element-pos morphrecs-indexed))
1303                                   (old-pos-record-lex-list
1304                                    (second old-pos-record))
1305                                   (old-pos-record-morphrecs
1306                                    (third old-pos-record)))
1307                              (if (not (member element-lex old-pos-record-lex-list))
1308                                  ;;This lex has not been recorded till now, so
1309                                  ;;add new morphrec to old-pos-record-morphrecs.
1310                                  (progn
1311                                    (setq number-of-different-morphemes (1+ number-of-different-morphemes))
1312                                    ;;<<CREATION OF MORPHREC
1313                                    (let* ((new-morphrec (make-morphrec element-lex element-pos)))
1314                                      (setq new-morphrec
1315                                            (put-morphrec-element new-morphrec 'segments (list all-segments-count)))
1316                                      (setq new-morphrec
1317                                            (put-morphrec-element new-morphrec 'hit-count 1))
1318                                      (setq new-pos-record
```

```
LINE #  TEXT
1319                                        (list element-pos
1320                                              (cons element-lex old-pos-record-lex-list)
1321                                              (cons
1322                                               new-morphrec
1323                                               old-pos-record-morphrecs)))))
1324
1325                                    ;;Else this lex has been recorded already,
1326                                    ;;so ADJOIN segment number onto SEGMENTS part of morphrec.
1327                                    ;;That is, cons it if it is not already there.
1328                                    ;;Thus segment number is entered for lex only once
1329                                    ;;even if it occurs more than once in that segment.
1330                                    (let* ((morphrec-old
1331                                            (assoc element-lex old-pos-record-morphrecs))
1332                                           ;;<<Increment total count.
1333                                           (new-hit-count (1+ (get-morphrec-element morphrec-old 'hit-count)))
1334                                           ;;<<UPDATING EXISTING MORPHREC
1335                                           (morphrec-new-temp
1336                                            ;;Note ADD.
1337                                            (add-morphrec-element morphrec-old 'segments all-segments-count))
1338                                           (morphrec-new
1339                                            (put-morphrec-element morphrec-new-temp 'hit-count new-hit-count))
1340                                           (new-pos-record-morphrecs
1341                                            (subst
1342                                             morphrec-new
1343                                             morphrec-old
1344                                             old-pos-record-morphrecs)))
1345                                      (setq new-pos-record
1346                                            (list element-pos
1347                                                  old-pos-record-lex-list
1348                                                  new-pos-record-morphrecs))))
1349                                (setq morphrecs-indexed
1350                                      (subst
1351                                       new-pos-record
1352                                       old-pos-record
1353                                       morphrecs-indexed)))))))
1354
1355          ;;If keyword arg RETURN-EXTRA-INFO? is non-nil, return list as shown below.
1356          ;;Else, for compatibility with previous uses of this fn,
1357          ;;return only MORPHRECS-INDEXED.
1358          (if return-extra-info?
1359              (list all-segments-count
1360                    corpus-length-in-morphemes
1361                    number-of-different-morphemes
1362                    morphrecs-indexed)
1363              morphrecs-indexed))))
1364
1365    ;;;============================================================================
1366    ;;;  II.B.5.  FINALIZING MORPHRECS-INDEXED, MAKING MORPHRECS-SORTED
1367    ;;;============================================================================
1368
1369    ;;;Receive morphrecs-indexed with nil in most fields of most morphrecs.
1370    ;;;Augment each MORPHREC and generate MORPHRECS-SORTED.
1371    ;;;<<Calculation of conditional probability, mutual information, etc.
1372    ;;;<<is localized in this function.
1373    (defun augment-morphrecs-indexed
1374        (morphrecs-indexed
1375         all-segments-count
1376         corpus-length-in-morphemes
1377         number-of-different-morphemes)
1378
1379      (let ((result nil)
1380            (morphrecs-sorted nil))
1381        ;;Augment every preliminary morphrec in corpus.
1382        (loop for pos in morphrecs-indexed do
1383              (let* ((pos-symbol (first pos))
1384                     (pos-morphemes (second pos))
1385                     (old-morphrecs (third pos))
1386                     (new-morphrecs
1387                      (loop for old-morphrec in old-morphrecs collect
1388                            (let* ((new-morphrec old-morphrec) ;initialize new-morphrec
1389                                   (morph-segments
1390                                    (get-morphrec-element old-morphrec 'segments)) ;segments this m appears in
1391                                   (morph-segments-count (length morph-segments)) ;count of segments this m appears in
1392                                   ;;Of all segments in corpus, what proportion contain this morph?
1393                                   (occur-in-segment-probability
1394                                    (round-to-n-digits
1395                                     (/ (float morph-segments-count)(float all-segments-count))
1396                                     *significant-digits*))
1397                                   (occur-in-segment-information
1398                                    (round-to-n-digits
1399                                     (log (/ (float 1) (float occur-in-segment-probability)) 2)
1400                                     *significant-digits*)) ;<<log base 2??????
1401                                   ;;Using 1+ probability smoothing.
1402                                   (occur-in-segment-probability-smoothed
1403                                    (round-to-n-digits
1404                                     (/ (+ (float morph-segments-count)(float 1))
1405                                        (+ (float 2)(float all-segments-count)))
1406                                     *significant-digits*))
1407                                   (hits-this-morpheme (get-morphrec-element old-morphrec 'hit-count))
1408                                   (morph-unigram-probability
1409                                    (round-to-n-digits
1410                                     (/ (float hits-this-morpheme)(float corpus-length-in-morphemes))
1411                                     *significant-digits*))
1412                                   (morph-unigram-information
1413                                    (round-to-n-digits
1414                                     (log (/ (float 1) (float morph-unigram-probability)) 2)
1415                                     *significant-digits*)) ;<<log base 2??????
1416                                   (morph-unigram-probability-smoothed
1417                                    (round-to-n-digits
1418                                     (/ (+ (float hits-this-morpheme)(float  1))
1419                                        (+ (float corpus-length-in-morphemes)
1420                                           (float number-of-different-morphemes)))
1421                                     *significant-digits*)))
1422
1423                              ;;<<AUGMENTATION OF MORPHREC
1424                              (setq new-morphrec
1425                                    (put-morphrec-element new-morphrec 'morph-segments-count morph-segments-count))
1426                              (setq new-morphrec
1427                                    (put-morphrec-element new-morphrec 'occur-in-segment-probability occur-in-segment-probability))
1428                              (setq new-morphrec
1429                                    (put-morphrec-element new-morphrec 'occur-in-segment-information occur-in-segment-information))
1430                              (setq new-morphrec
1431                                    (put-morphrec-element new-morphrec 'occur-in-segment-probability-smoothed occur-in-segment-probability-smoothe
                                      d))
1432                              (setq new-morphrec
1433                                    (put-morphrec-element new-morphrec 'morph-unigram-probability morph-unigram-probability))
1434                              (setq new-morphrec
1435                                    (put-morphrec-element new-morphrec 'morph-unigram-information morph-unigram-information))
1436                              (setq new-morphrec
1437                                    (put-morphrec-element new-morphrec 'morph-unigram-probability-smoothed morph-unigram-probability-smoothed)))))
```

```
1438                    (new-pos (list pos-symbol pos-morphemes new-morphrecs)))
1439
1440              ;;Add augmented pos record to result.
1441              (setq result (cons new-pos result))
1442
1443              ;;Add augmented hit records to MORPHRECS-SORTED.
1444              (setq morphrecs-sorted
1445                    (append new-morphrecs morphrecs-sorted))))
1446
1447      ;;;After loop, return both augmented MORPHRECS-INDEXED and MORPHRECS-SORTED.
1448      (list (reverse result)(reverse morphrecs-sorted))))
1449
1450  ;;;============================================================================================
1451  ;;;  II.B.6.  SORTING MORPHRECS-INDEXED AND MORPHRECS-SORTED
1452  ;;;============================================================================================
1453
1454  (defun resort-morphrecs-indexed
1455      (morphrecs-indexed
1456       &key (resort-by 'hit-count))
1457    (let ((result nil))
1458      (loop for pos in morphrecs-indexed do
1459            (let* ((morphrecs (third pos))
1460                   (morphrecs-sorted
1461                    (sort-morphrecs morphrecs resort-by))
1462                   (new-morph-list
1463                    (loop for record in morphrecs-sorted collect (first record)))
1464                   (new-pos (list (first pos) new-morph-list morphrecs-sorted)))
1465              (setq result (cons new-pos result))))
1466      (reverse result)))
1467
1468  (defun sort-morphrecs (morphrecs element-name)
1469    (let ((position (get-morphrec-element-position element-name)))
1470      ;;LAMBDA means "test whether list-1 is greater than list-2
1471      ;;with respect to the elements at nth position"
1472      (sort morphrecs
1473            #'(lambda (list1 list2)
1474                (if (not (numberp (nth position list1)))
1475                    (error "can't sort: pointer to non-numerical element in morphrec")
1476                    (> (nth position list1)(nth position list2)))))))
1477
1478  ;;;============================================================================================
1479  ;;;  II.B.7.  QUERYING MORPHRECS-SORTED
1480  ;;;============================================================================================
1481
1482  ;;;Collects list of e.g. MORPH-UNIGRAM-PROBABILITY values over all morph records.
1483  (defun get-all-morphrec-elements
1484      (element-name morphrecs-sorted)
1485    (loop for morphrec in morphrecs-sorted collect
1486          (get-morphrec-element morphrec element-name)))
1487
1488  ;;;Morph records contain pre-computed pairs like MORPH-UNIGRAM-PROBABILITY and MORPH-UNIGRAM-INFORMATION.
1489  ;;;This function gets the SUM OF THE PRODUCTS of these pairs OVER ALL MORPHS.
1490  (defun get-morphrec-element-entropy (probability-element information-element morphrecs-sorted)
1491    (let* ((probabilities (get-all-morphrec-elements probability-element morphrecs-sorted))
1492           (informations (get-all-morphrec-elements information-element morphrecs-sorted))
1493           (products
1494            (loop for probability in probabilities and information in informations collect
1495                  (* probability information)))
1496           (sum 0.0))
1497      (loop for product in products do
1498            (setq sum (+ sum product)))
1499      sum))
1500
1501  ;;;============================================================================================
1502  ;;;  II.B.8.  QUERYING MORPHRECS-INDEXED: INCLUDES FINDING WINDOW MATES
1503  ;;;============================================================================================
1504
1505  (defvar *MORPHRECS-INDEXED* nil)
1506
1507  (defun load-morphrecs-indexed
1508      (morphrecs-indexed-file)
1509    (setq *MORPHRECS-INDEXED*
1510          (read-from-file morphrecs-indexed-file)))
1511
1512  (defun get-morphrecs-with-files
1513      (lex directory corpus-prefix &key pos (db-pos-display 'english))
1514    (let* ((morphrecs-indexed-file (format nil "~A~A.morphrecs.indexed" directory corpus-prefix)))
1515      (if (null *morphrecs-indexed*)
1516          (progn
1517            (format t "~%Loading morphrecs-indexed from file ~a." morphrecs-indexed-file)
1518            (load-morphrecs-indexed morphrecs-indexed-file)))
1519      (get-morphrecs lex *morphrecs-indexed* :pos pos :db-pos-display db-pos-display)))
1520
1521  ;;;Returns list of MORPHRECS.
1522  ;;;Multiple morphrecs should be contained only if no POS constraint.
1523  ;;;is given LEX has multiple parts-of-speech.
1524  (defun get-morphrecs (lex morphrecs-indexed &key pos (db-pos-display 'english))
1525    (let ((morphrecs nil)
1526          (pos-record
1527           (if pos (assoc pos morphrecs-indexed))))
1528      (if pos
1529          (setq morphrecs (list (assoc lex (third pos-record))))
1530
1531          ;;Else no pos was specified, so we have to try all pos.
1532          (loop for this-pos in *pos-table* do
1533                (let ((this-pos-symbol
1534                       (case db-pos-display
1535                         ((number) (first this-pos))
1536                         ((english)(fourth this-pos))
1537                         ((japanese (second this-pos))))))
1538                  (setq pos-record (assoc this-pos-symbol morphrecs-indexed))
1539                  (let ((findings-for-this-pos (assoc lex (third pos-record))))
1540                    (if findings-for-this-pos
1541                        (setq morphrecs (cons findings-for-this-pos morphrecs)))))))
1542      morphrecs))
1543
1544  (defun get-morphrecs-for-pos (pos morphrecs-indexed)
1545    (third (assoc pos morphrecs-indexed :test #'equal)))
1546
1547  (defun get-morph-hit-count (lex pos morphrecs-indexed &key (db-pos-display 'english))
1548    (let* ((morphrecs (get-morphrecs lex morphrecs-indexed :pos pos :db-pos-display db-pos-display))
1549           (morphrec (first morphrecs))
1550           (hit-count (get-morphrec-element morphrec 'hit-count)))
1551      (if hit-count hit-count 0)))
1552
1553  (defun get-morph-segments (lex pos morphrecs-indexed &key (db-pos-display 'english))
1554    (let* ((morphrecs (get-morphrecs lex morphrecs-indexed :pos pos :db-pos-display db-pos-display))
1555           (morphrec (first morphrecs))
1556           (segments (get-morphrec-element morphrec 'segments)))
1557      (if segments segments nil)))
```

LINE # TEXT

```
1558
1559  (defun morph-occurs-in-segment? (lex pos segment morphrecs-indexed &key (db-pos-display 'english))
1560    (member segment (get-morph-segments lex pos morphrecs-indexed :db-pos-display db-pos-display)))
1561
1562  ;;;Also used as a predicate to test whether there are ANY such occurrences.
1563  (defun get-morph-occurrences-within-n-segments
1564      (lex pos reference-segment n morphrecs-indexed discourse-final-segments &key (db-pos-display 'english))
1565    (ignore db-pos-display)
1566    (let ((result nil)
1567          (morph-segments
1568           (get-morph-segments lex pos morphrecs-indexed :db-pos-display 'english)))
1569      (loop for segment in morph-segments do
1570            (if (and (>= segment reference-segment)
1571                     (<= (- segment reference-segment) n)
1572                     (in-same-discourse? segment reference-segment discourse-final-segments))
1573                (setq result (cons segment result))))
1574      (reverse result)))
1575
1576  ;;;Determines whether segment1 and segment2 are in same discourse.
1577  ;;;Is there a member of DISCOURSE-FINAL-SEGMENTS which is >= seg1 and < seg2?
1578  (defun in-same-discourse? (segment1 segment2 discourse-final-segments)
1579    (let ((found nil)
1580          (result t))
1581      (loop for discourse-final-segment in discourse-final-segments until found do
1582            (if (and (>= discourse-final-segment segment1)
1583                     (< discourse-final-segment segment2)) ;less than, not equal
1584                (progn (setq found t)
1585                       (setq result nil))))
1586      result))
1587
1588  ;;;============================================================================
1589  ;;;  II.B.9.  DEFINING MORPHREC-DIFFREC
1590  ;;;============================================================================
1591
1592  ;;;A MORPHREC-DIFFREC is a record of the difference between two corpora for a given morph.
1593  (defvar *morphrec-diffrec-elements*
1594    '((lex 0)
1595      (pos 1)
1596      (element-1 2)
1597      (element-2 3)
1598      (difference 4)
1599      (count-1 5)  ;morph-1 hit-count corpus-1
1600      (count-2 6)  ;morph-1 hit-count corpus-2
1601      (count-sum 7)))
1602
1603  (defun make-morphrec-diffrec (lex pos element-1 element-2 difference count-1 count-2 count-sum)
1604    (let ((morphrec-diffrec
1605           (loop for element in *morphrec-diffrec-elements* collect nil)))
1606      (setq morphrec-diffrec (put-morphrec-diffrec-element morphrec-diffrec 'lex lex))
1607      (setq morphrec-diffrec (put-morphrec-diffrec-element morphrec-diffrec 'pos pos))
1608      (setq morphrec-diffrec (put-morphrec-diffrec-element morphrec-diffrec 'element-1 element-1))
1609      (setq morphrec-diffrec (put-morphrec-diffrec-element morphrec-diffrec 'element-2 element-2))
1610      (setq morphrec-diffrec (put-morphrec-diffrec-element morphrec-diffrec 'difference difference))
1611      (setq morphrec-diffrec (put-morphrec-diffrec-element morphrec-diffrec 'count-1 count-1))
1612      (setq morphrec-diffrec (put-morphrec-diffrec-element morphrec-diffrec 'count-2 count-2))
1613      (setq morphrec-diffrec (put-morphrec-diffrec-element morphrec-diffrec 'count-sum count-sum))
1614      morphrec-diffrec))
1615
1616  ;;;<<first position is zero, not 1!!
1617  (defun get-morphrec-diffrec-element-position (element-name)
1618    (second
1619     (assoc element-name *morphrec-diffrec-elements*)))
1620
1621  (defun get-morphrec-diffrec-element (morphrec-diffrec element-name)
1622    (let ((position (get-morphrec-diffrec-element-position element-name)))
1623      (nth position morphrec-diffrec)))
1624
1625  (defun put-morphrec-diffrec-element (morphrec-diffrec element-name value)
1626    (let ((morphrec-diffrec-copy (copy-list morphrec-diffrec))
1627          (position (get-morphrec-diffrec-element-position element-name)))
1628      (setf (nth position morphrec-diffrec-copy) value)
1629      morphrec-diffrec-copy))
1630
1631  (defun add-morphrec-diffrec-element (morphrec-diffrec element-name value)
1632    (let ((morphrec-diffrec-copy (copy-list morphrec-diffrec))
1633          (old-value (get-morphrec-diffrec-element morphrec-diffrec element-name))
1634          (new-value nil)
1635          (position (get-morphrec-diffrec-element-position element-name)))
1636      (if (not (listp old-value))
1637          (error "Old value not a list.")
1638          (progn (setq new-value (adjoin value old-value :test #'equal))
1639                 (setf (nth position morphrec-diffrec-copy) new-value)))
1640      morphrec-diffrec-copy))
1641
1642  (defun sort-morphrec-diffrecs (morphrec-diffrecs element-name)
1643    (let ((position (get-morphrec-diffrec-element-position element-name)))
1644      ;;LAMBDA means "test whether list-1 is greater than list-2
1645      ;;with respect to the elements at nth position"
1646      (sort morphrec-diffrecs
1647            #'(lambda (list1 list2)
1648                (if (not (numberp (nth position list1)))
1649                    (error "can't sort: pointer to non-numerical element in rec")
1650                    (> (nth position list1)(nth position list2)))))))
1651
1652  ;;;============================================================================
1653  ;;;  II.B.10.  COMPARING MORPHRECS-SORTED
1654  ;;;============================================================================
1655
1656  ;;;Compare morphrecs-sorted for two corpora.
1657  ;;;Includes use of KULLBACK-LEIBLER formula for "RELATIVE ENTROPY".
1658  ;;;Assumes MORPHRECS-SORTED-FILES have been presorted as desired.
1659  (defun compare-morphrecs-sorted
1660      (directory corpus-prefix-1 corpus-prefix-2 &key (element 'morph-unigram-probability))
1661
1662    (let* ((morphrecs-indexed-file-1 (format nil "~A~A.morphrecs.indexed"  directory corpus-prefix-1))
1663           (morphrecs-sorted-file-1 (format nil "~A~A.morphrecs.sorted"  directory corpus-prefix-1))
1664           (morphrecs-indexed-file-2 (format nil "~A~A.morphrecs.indexed"  directory corpus-prefix-2))
1665           (morphrecs-sorted-file-2 (format nil "~A~A.morphrecs.sorted"  directory corpus-prefix-2))
1666           (output-file (format nil "~A~A.vs.~A.morphrecs.comparison"  directory corpus-prefix-1 corpus-prefix-2))
1667           (morphrecs-indexed-1 (read-from-file morphrecs-indexed-file-1))
1668           (morphrecs-indexed-2 (read-from-file morphrecs-indexed-file-2))
1669           (morphrecs-sorted-1 (read-from-file morphrecs-sorted-file-1))
1670           (morphs-1
1671            (loop for morph in morphrecs-sorted-1 collect
1672                  (list
1673                   (get-morphrec-element morph 'lex)
1674                   (get-morphrec-element morph 'pos))))
1675           (morphrecs-sorted-2 (read-from-file morphrecs-sorted-file-2))
1676           (morphs-2
1677            (loop for morph in morphrecs-sorted-2 collect
```

LINE # TEXT

```lisp
               (list
                (get-morphrec-element morph 'lex)
                (get-morphrec-element morph 'pos))))
         (in-both (intersection morphs-1 morphs-2 :test #'equal))
         (in-both-count (length in-both))
         (in-morphs-1-but-not-in-morphs-2 (set-difference morphs-1 morphs-2 :test #'equal))
         (in-morphs-2-but-not-in-morphs-1 (set-difference morphs-2 morphs-1 :test #'equal))
         (in-both-difference-records nil)
         (summed-in-both-differences 0.0)
         (average-in-both-differences nil)
         (average-morphs-1-element nil)
         (average-morphs-2-element nil)
         (summed-morphs-1-elements 0.0)
         (summed-morphs-2-elements 0.0)
         (relative-entropy 0.0))

    (if output-file
        (if (probe-file output-file)
            (delete-file output-file)))

    ;;Loop through MORPHRECS-SORTED-FILE-1, the training corpus,
    ;;so that this ordering will be preserved.
    ;;If record-1 is for a morph which is in both corpora, go ahead.
    (loop for record-1 in morphrecs-sorted-1 do
          (let ((lex (first record-1))
                (pos (second record-1)))
            (if (member (list lex pos) in-both :test #'equal)
                (let* ((record-2 (first (get-morphrecs lex morphrecs-indexed-2 :pos pos)))
                       ;;ELEMENT is e.g. morph-unigram-probability
                       (element-1 (get-morphrec-element record-1 element))
                       (element-2 (get-morphrec-element record-2 element))
                       (difference
                        (round-to-n-digits
                         (abs (- element-1 element-2)) *significant-digits*)))

                  ;;Make a new DIFFERENCE-RECORD, showing the respective values
                  ;;for the factor of interest in the two corpora
                  ;;and the difference between them.
                  (let* ((morphrec-1 (first (get-morphrecs lex morphrecs-indexed-1 :pos pos)))
                         (morphrec-2 (first (get-morphrecs lex morphrecs-indexed-2 :pos pos)))
                         (count-1 (get-morphrec-element morphrec-1 'hit-count))
                         (count-2 (get-morphrec-element morphrec-2 'hit-count))
                         (count-sum (+ count-1 count-2))
                         (new-morphrec-diffrec
                          (make-morphrec-diffrec
                           lex pos element-1 element-2 difference count-1 count-2 count-sum)))

                    (setq in-both-difference-records
                          (cons
                           new-morphrec-diffrec
                           in-both-difference-records)))

                  ;;Maintain running sums.
                  (setq summed-in-both-differences (+ summed-in-both-differences difference))
                  (setq summed-morphs-1-elements (+ summed-morphs-1-elements element-1))
                  (setq summed-morphs-2-elements (+ summed-morphs-2-elements element-2))

                  ;;<<Kullback-Leibler formula for "relative entropy".
                  ;;Sum over all elements: Pi log 1/Qi
                  (if (equal element 'morph-unigram-probability)
                      ;;Pi = element-2 (test corpus), Qi = element-1 (training corpus)
                      (let ((summed (* element-2 (log (/ (float element-2)(float element-1)) 2))))
                        (setq relative-entropy (+ relative-entropy summed)))))))))

    (setq in-both-difference-records (reverse in-both-difference-records))

    (setq relative-entropy (round-to-n-digits relative-entropy *significant-digits*))

    ;;Get averages if any records are shared.
    ;;If not, averages remain nil.
    (if in-both-count
        (progn
          (setq average-in-both-differences
                (round-to-n-digits
                 (/ (float summed-in-both-differences) (float in-both-count)) *significant-digits*))
          (setq average-morphs-1-element
                (round-to-n-digits
                 (/ (float summed-morphs-1-elements) (float in-both-count)) *significant-digits*))
          (setq average-morphs-2-element
                (round-to-n-digits
                 (/ (float summed-morphs-2-elements) (float in-both-count)) *significant-digits*))))

    ;;PRINTING STARTS
    (smart-format output-file "~%;;;~A and ~A ~%;;;compared in terms of ~s~3%"
                  morphrecs-sorted-file-1 morphrecs-sorted-file-2 element)

    (if (equal element 'morph-unigram-probability)
        (smart-format output-file "~2%(relative-entropy ~s)"
                      relative-entropy))
    (smart-format output-file "~2%(average-~s-corpus-1 ~s)" element average-morphs-1-element)
    (smart-format output-file "~2%(average-~s-corpus-2 ~s)" element average-morphs-2-element)
    (smart-format output-file "~2%(average-~s-difference  ~s)"
                  element average-in-both-differences)

    (smart-format output-file "~2%(in-both-morphs-count ~s)"
                  (length in-both-difference-records))
    (smart-format output-file "~2%(difference-records-corpus-1-order ~s)" in-both-difference-records)

    (smart-format output-file "~2%(in-corpus-1-but-not-corpus-2-count ~s)"
                  (length in-morphs-1-but-not-in-morphs-2))
    (smart-format output-file "~2%(in-corpus-1-but-not-corpus-2 ~s)" in-morphs-1-but-not-in-morphs-2)

    (smart-format output-file "~2%(in-corpus-2-but-not-corpus-1-count ~s)"
                  (length in-morphs-2-but-not-in-morphs-1))
    (smart-format output-file "~2%(in-corpus-2-but-not-corpus-1 ~s)" in-morphs-2-but-not-in-morphs-1)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  II.C. MORPH-COOCRECS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;========================================================================
;;;  II.C.1.  DEFINING MORPH-COOCREC DATASTRUCTURE
;;;========================================================================

(defvar *morph-coocrec-elements*
  '((n 0)
    (pos-1 1)
    (lex-1 2)
    (pos-2 3)
    (lex-2 4)
```

```
LINE #  TEXT
1798      (morph-1-segments-with-2-in-range-count 5)
1799      (morph-1-segments-count 6)
1800      (conditional-probability 7)
1801      (mutual-information 8)
1802      (conditional-probability-smoothed 9)
1803      (mutual-information-smoothed 10)))
1804
1805  ;;;never called
1806  ;(defun make-morph-coocrec (n pos-1 lex-1 pos-2 lex-2)
1807  ;  (let ((morph-coocrec
1808  ;         (loop for element in *morph-coocrec-elements* collect nil)))
1809  ;    (setq morph-coocrec (put-morph-coocrec-element morph-coocrec 'n n))
1810  ;    (setq morph-coocrec (put-morph-coocrec-element morph-coocrec 'pos-1 pos-1))
1811  ;    (setq morph-coocrec (put-morph-coocrec-element morph-coocrec 'lex-1 lex-1))
1812  ;    (setq morph-coocrec (put-morph-coocrec-element morph-coocrec 'pos-2 pos-2))
1813  ;    (setq morph-coocrec (put-morph-coocrec-element morph-coocrec 'lex-2 lex-2))
1814  ;    morph-coocrec))
1815
1816  (defun get-morph-coocrec-element-position (element-name)
1817    (second
1818     (assoc element-name *morph-coocrec-elements*)))
1819
1820  (defun get-morph-coocrec-element (morph-coocrec element-name)
1821    (let ((position (get-morph-coocrec-element-position element-name)))
1822      (nth position morph-coocrec)))
1823
1824  (defun put-morph-coocrec-element (morph-coocrec element-name value)
1825    (let ((morph-coocrec-copy (copy-list morph-coocrec))
1826          (position (get-morph-coocrec-element-position element-name)))
1827      (setf (nth position morph-coocrec-copy) value)
1828      morph-coocrec-copy))
1829
1830  ;;;============================================================================
1831  ;;;  II.C.2.  MAKING MORPH-COOCRECS-INDEXED
1832  ;;;============================================================================
1833
1834  (defun make-morph-coocrecs-indexed-with-files
1835      (n-range pos-1-list pos-2-list directory corpus-prefix
1836               &key (db-pos-display 'english)(threshold 0.1)(threshold-key 'conditional-probability))
1837    (let* ((segments-file (format nil "~A~A.segments" directory
1838                                  corpus-prefix))
1839           (segments-info (read-from-file segments-file))
1840           (all-segments-count (length (first segments-info)))
1841           (discourse-final-segments (second segments-info))
1842           (morphrecs-indexed-file (format nil "~A~A.morphrecs.indexed" directory corpus-prefix))
1843           (morphrecs-indexed (read-from-file morphrecs-indexed-file))
1844           (morph-coocrecs-indexed
1845            (make-morph-coocrecs-indexed
1846             n-range pos-1-list pos-2-list morphrecs-indexed discourse-final-segments all-segments-count
1847             :db-pos-display db-pos-display :threshold threshold :threshold-key threshold-key))
1848          (morph-coocrecs-indexed-file (format nil "~A~A.morph.coocrecs.indexed" directory corpus-prefix)))
1849      (format t "Writing to ~S~%" morph-coocrecs-indexed-file)
1850      (write-to-file-no-pp morph-coocrecs-indexed morph-coocrecs-indexed-file)))
1851
1852  ;;;Makes morph-coocrecs-indexed, given MORPHRECS-INDEXED as input.
1853  ;;;Dynamically filters records below a threshold.
1854  (defun make-morph-coocrecs-indexed
1855      (n-range pos-1-list pos-2-list morphrecs-indexed discourse-final-segments all-segments-count
1856               &key (db-pos-display 'english)(threshold 0.1)(threshold-key 'conditional-probability))
1857    (loop for n from (first n-range) to (second n-range) collect
1858          (progn (format t "Handling n = ~S~%" n)
1859                 (list n
1860                       (loop for pos-1 in pos-1-list collect
1861                             (progn (format t "  Handling pos-1 = ~S~%" pos-1)
1862                                    (list pos-1
1863                                          (loop for lex-1 in (second (assoc pos-1 morphrecs-indexed)) collect
1864                                                (list lex-1
1865                                                      (loop for pos-2 in pos-2-list collect
1866                                                            (list pos-2
1867                                                                  ;;<<note use of IT loop construct see Steele 739
1868                                                                  (loop for lex-2 in
1869                                                                        (second (assoc pos-2 morphrecs-indexed))
1870                                                                        if
1871                                                                        (and
1872                                                                         ;;Ignore co-occurrence of morph with itself.
1873                                                                         (not
1874                                                                          (and (equal pos-1 pos-2)
1875                                                                               (equal lex-1 lex-2)))
1876                                                                         (make-morph-coocrec-tail
1877                                                                          lex-1 pos-1 lex-2 pos-2
1878                                                                          n morphrecs-indexed
1879                                                                          discourse-final-segments
1880                                                                          all-segments-count
1881                                                                          :db-pos-display db-pos-display
1882                                                                          :threshold threshold
1883                                                                          :threshold-key threshold-key))
1884                                                                        collect it))))))))))))
1885
1886  ;;;============================================================================
1887  ;;;  II.C.3.  MAKE-MORPH-COOCREC-TAIL
1888  ;;;============================================================================
1889
1890  ;;;Given morph-1 in segment S1, compute the conditional probability and mutual information
1891  ;;;that morph-2 will occur within n segments.
1892  ;;;Also calculates smoothed versions of both results.
1893
1894  ;;;Returns the tail of a MORPH-COOCREC -- not a complete morph-coocrec --
1895  ;;;to be entered as terminal of MORPH-COOCRECS-INDEXED:
1896  ;;;(list lex-2
1897  ;;;      morph-1-segments-with-2-in-range-count
1898  ;;;      morph-1-segments-count
1899  ;;;      conditional-probability
1900  ;;;      mutual-information
1901  ;;;      conditional-probability-smoothed
1902  ;;;      mutual-information-smoothed )
1903
1904  ;;;Pr (b|a) = Pr (a, b)/ Pr (a)
1905  ;;;conditional probability that morph-2 will occur within n segments of segment S,
1906  ;;;given that morph-1 appears in segment S.
1907
1908  ;;;Pr (a) : probability that morph-1 occurs (at least once) in segment S.
1909  ;;;LOCAL VARIABLE: MORPH-1-OCCUR-IN-SEGMENT-PROBABILITY(-SMOOTHED)
1910  ;;;In other words, probability that any segment Si fits this description:
1911  ;;;morph-1 occurs in Si. (More formally, "morph-1 is element of Si".)
1912  ;;;CALCULATION: Count (segments in which morph-1 occurs at least once)/ Count (segments in corpus)
1913
1914  ;;;Pr (b) : probability that morph-2 will occur at least once
1915  ;;;within n segments of segment Si (in dialogue D)
1916  ;;; -- that is, within a window whose first segment is Si.
1917  ;;;In other words, probability for any segment Si that:
```

```
1918 ;;;morph-2 occurs in Si, or in its successor Si+1, Si+2, .... up to Si+n.
1919 ;;;(More formally, that "morph-2 is element of Si union Si+1 union .... Si + n".)
1920 ;;;LOCAL VARIABLE: MORPH-2-OCCUR-IN-WINDOW-PROBABILITY
1921 ;;;CALCULATION: MORPH-2-OCCUR-IN-SEGMENT-PROBABILITY * (n + 1)
1922 ;;;Note that Pr(b) is also included in local variable corresponding to Pr(a, b):
1923 ;;;MORPH-1-OCCUR-IN-SEGMENT-WITH-2-IN-RANGE-PROBABILITY(-SMOOTHED)
1924
1925 ;;;Pr(a, b): probability that morph-1 appears (at least once) in segment Si AND
1926 ;;;morph-2 will occur at least once within n segments of segment Si (in dialogue D).
1927 ;;;In other words, probability for any segment Si that:
1928 ;;;morph-1 occurs in Si AND
1929 ;;;morph-2 occurs in Si, or in its successor Si+1, Si+2, .... up to Si+n.
1930 ;;;(More formally, that "morph-1 is an element of Si and morph-2 is element of Si union S2 union .... Si + n".)
1931 ;;;LOCAL VARIABLE: MORPH-1-OCCUR-IN-SEGMENT-WITH-2-IN-RANGE-PROBABILITY(-SMOOTHED)
1932 ;;;CALCULATION: MORPH-1-SEGMENTS-WITH-2-IN-RANGE-COUNT/ALL-SEGMENTS-COUNT
1933
1934 ;;;Returns a record only when conditional-probability >= THRESHOLD
1935 (defun make-morph-coocrec-tail
1936     (lex-1 pos-1 lex-2 pos-2 n morphrecs-indexed discourse-final-segments all-segments-count
1937         &key (db-pos-display 'english)(threshold 0.1)(threshold-key 'conditional-probability))
1938   (let* ((morphrec-1 (first (get-morphrecs lex-1 morphrecs-indexed :pos pos-1)))
1939          ;;SEGMENTS CONTAINING MORPH-1 AT LEAST ONCE
1940          (morph-1-segments
1941           (get-morphrec-element morphrec-1 'segments))
1942          ;;NUMBER OF SEGMENTS CONTAINING MORPH-1 AT LEAST ONCE
1943          (morph-1-segments-count (length morph-1-segments))
1944
1945          ;;PROBABILITY THAT MORPH-1 OCCURS IN ANY GIVEN SEGMENT
1946          ;;Pr(a) = MORPH-1-SEGMENTS-COUNT/ALL-SEGMENTS-COUNT
1947          (morph-1-occur-in-segment-probability
1948           (get-morphrec-element morphrec-1 'occur-in-segment-probability))
1949
1950          ;;SMOOTHED PROBABILITY THAT MORPH-1 OCCURS IN ANY GIVEN SEGMENT
1951          ;;Pr(a) smoothed
1952          (morph-1-occur-in-segment-probability-smoothed
1953           (get-morphrec-element morphrec-1 'occur-in-segment-probability-smoothed))
1954
1955          (morph-1-segments-with-2-in-range-count 0)
1956          ;;Pr(a and b)
1957          (morph-1-occur-in-segment-with-2-in-range-probability 0)
1958          ;;Pr(a and b) smoothed
1959          (morph-1-occur-in-segment-with-2-in-range-probability-smoothed 0)
1960
1961          (morphrec-2 (first (get-morphrecs lex-2 morphrecs-indexed :pos pos-2)))
1962
1963          ;;PROBABILITY THAT MORPH-2 OCCURS IN ANY GIVEN SEGMENT
1964          ;; = MORPH-2-SEGMENTS-COUNT/ALL-SEGMENTS-COUNT
1965          (morph-2-occur-in-segment-probability (get-morphrec-element morphrec-2 'occur-in-segment-probability))
1966          (morph-2-occur-in-segment-probability-smoothed (get-morphrec-element morphrec-2 'occur-in-segment-probability-smoothed))
1967
1968          ;;PROBABILITY THAT MORPH-2 OCCURS IN ANY GIVEN WINDOW OF WIDTH N PLUS 1.
1969          ;;Pr(b)
1970          (morph-2-occur-in-window-probability (* (float (+ n 1)) morph-2-occur-in-segment-probability))
1971          (morph-2-occur-in-window-probability-smoothed (* (float (+ n 1)) morph-2-occur-in-segment-probability-smoothed)))
1972
1973     ;;Get MORPH-1-SEGMENTS-WITH-2-IN-RANGE-COUNT
1974     (loop for morph-1-segment in morph-1-segments do
1975           (if
1976            ;;Use this fn to test, for each segment Si where morph-1 occurs,
1977            ;;whether morph-2 occurs in any segment within range n of Si.
1978            ;;If so, increment MORPH-1-SEGMENTS-WITH-2-IN-RANGE-COUNT.
1979            (get-morph-occurrences-within-n-segments
1980             lex-2 pos-2 morph-1-segment n morphrecs-indexed discourse-final-segments :db-pos-display db-pos-display)
1981            (setq morph-1-segments-with-2-in-range-count (1+ morph-1-segments-with-2-in-range-count))))
1982
1983     ;;Get MORPH-1-OCCUR-IN-SEGMENT-WITH-2-IN-RANGE-PROBABILITY
1984     ;;Pr(a and b)
1985     (setq morph-1-occur-in-segment-with-2-in-range-probability
1986           (round-to-n-digits
1987            (/ (float morph-1-segments-with-2-in-range-count)(float all-segments-count)) *significant-digits*))
1988
1989     ;;Get MORPH-1-OCCUR-IN-SEGMENT-WITH-2-IN-RANGE-PROBABILITY-SMOOTHED
1990     ;;Using 1+ smoothing.
1991     ;;Pr(a and b) smoothed
1992     (setq morph-1-occur-in-segment-with-2-in-range-probability-smoothed
1993           (round-to-n-digits
1994            (/ (+ (float morph-1-segments-with-2-in-range-count)(float 1))
1995               (+ (float 2)(float all-segments-count))) *significant-digits*))
1996
1997     ;;Compute conditional-probability that morph-2 will occur within n segments of segment S,
1998     ;;given that morph-1 occurs in S.
1999     ;;Return record iff conditional-probability >= threshold. Else return nil.
2000     ;;Pr (b given a) = Pr (a and b)/ Pr (a)
2001     (let* ((conditional-probability
2002             (if (or (zerop morph-1-occur-in-segment-with-2-in-range-probability)
2003                     (zerop morph-1-occur-in-segment-probability))
2004                 ;;<<Giving zero in case denominator is 0.0.
2005                 0.0
2006                 (round-to-n-digits
2007                  (/ (float morph-1-occur-in-segment-with-2-in-range-probability)
2008                     (float morph-1-occur-in-segment-probability)) *significant-digits*)))
2009
2010            ;;log P(b given a) / P(b)
2011            (mutual-information
2012             (if (or (zerop conditional-probability)
2013                     (zerop morph-2-occur-in-window-probability))
2014                 ;;<<Giving zero in case denominator is 0.0.
2015                 0.0
2016                 (round-to-n-digits
2017                  (log (/ (float conditional-probability)
2018                          (float morph-2-occur-in-window-probability)) 2) *significant-digits*)))
2019
2020            ;;As above, but with smoothed nominators and divisors.
2021            (conditional-probability-smoothed
2022             (if (or (zerop morph-1-occur-in-segment-with-2-in-range-probability-smoothed)
2023                     (zerop morph-1-occur-in-segment-probability-smoothed))
2024                 ;;<<Giving zero in case denominator is 0.0.
2025                 0.0
2026                 (round-to-n-digits
2027                  (/ (float morph-1-occur-in-segment-with-2-in-range-probability-smoothed)
2028                     (float morph-1-occur-in-segment-probability-smoothed)) *significant-digits*)))
2029
2030            ;;As above, but with smoothed nominators and divisors.
2031            (mutual-information-smoothed
2032             (if (or (zerop conditional-probability-smoothed)
2033                     (zerop morph-2-occur-in-window-probability-smoothed))
2034                 ;;<<Giving zero in case denominator is 0.0.
2035                 0.0
2036                 (round-to-n-digits
2037                  (log (/ (float conditional-probability-smoothed)
```

```
2038                          (float morph-2-occur-in-window-probability-smoothed)) 2) *significant-digits*)))
2039            (threshold-key-value
2040             (case threshold-key
2041               ((conditional-probability)
2042                conditional-probability)
2043               ((conditional-probability-smoothed)
2044                conditional-probability-smoothed)
2045               ((mutual-information)
2046                mutual-information)
2047               ((mutual-information-smoothed)
2048                mutual-information-smoothed))))
2049
2050       ;;Returns list to be entered as terminal of MORPH-COOCRECS-INDEXED;
2051       ;;NOT a complete MORPH-COOCREC.
2052       (if (>= threshold-key-value threshold)
2053           (list lex-2
2054                 morph-1-segments-with-2-in-range-count
2055                 morph-1-segments-count
2056                 conditional-probability
2057                 mutual-information
2058                 conditional-probability-smoothed
2059                 mutual-information-smoothed)))))
2060
2061  ;;;============================================================================
2062  ;;;  II.C.4.  QUERYING MORPH-COOCRECS-INDEXED
2063  ;;;============================================================================
2064
2065  (defvar *MORPH-COOCRECS-INDEXED* nil)
2066
2067  (defun load-morph-coocrecs-indexed
2068      (morph-coocrecs-indexed-file)
2069    (setq *MORPH-COOCRECS-INDEXED*
2070          (read-from-file morph-coocrecs-indexed-file))
2071    (format t "~%LOADING COMPLETED: ~a" morph-coocrecs-indexed-file))
2072
2073  ;;;Gets from MORPH-COOCRECS-INDEXED the conditional probability of occurrence within N segments
2074  ;;;for specified MORPH-1 and MORPH-2.
2075  (defun get-morph-conditional-probability
2076      (n pos-1 lex-1 pos-2 lex-2 morph-coocrecs-indexed &key (db-pos-display 'english))
2077    (get-morph-coocrec-element
2078     (make-and-get-morph-coocrec
2079      n lex-1 pos-1 lex-2 pos-2 morph-coocrecs-indexed :db-pos-display db-pos-display)
2080     'conditional-probability))
2081
2082  ;;;Makes and gets (fetches) MORPH-COOCREC.
2083  ;;;Fetches partial morph-coocrec, which is terminal of MORPH-COOCRECS-INDEXED.
2084  ;;;Completes it, creating and returning full MORPH-COOCREC.
2085  ;;;(2 COMMON-NOUN 階段 COMMON-NOUN 右 3 3 1.0 0.0 0.8)
2086  ;;;<<Note that MAKE-MORPH-COOCREC is not called to avoid clumsy copying.
2087  ;;;<<It is provided only for completeness.
2088  (defun make-and-get-morph-coocrec
2089      (n lex-1 pos-1 lex-2 pos-2 morph-coocrecs-indexed &key (db-pos-display 'english))
2090    (ignore db-pos-display)
2091    (let* ((n-group (second (assoc n morph-coocrecs-indexed)))
2092           (pos-1-group (second (assoc pos-1 n-group)))
2093           (lex-1-group (second (assoc lex-1 pos-1-group)))
2094           (pos-2-group (second (assoc pos-2 lex-1-group)))
2095           (lex-2-record (assoc lex-2 pos-2-group)))
2096      (if lex-2-record
2097          ;;<<CREATION OF MORPH-COOCREC
2098          ;;<<Avoiding MAKE-MORPH-COOCREC to avoid clumsy copying.
2099          (append (list n pos-1 lex-1 pos-2) lex-2-record)
2100          (append (list n pos-1 lex-1 pos-2 lex-2) '(0.0 0.0 0.0 0.0 0.0 0.0)))))
2101
2102  ;;;============================================================================
2103  ;;;  II.C.5.  MORPH-WINDOW-MATE QUERY FUNCTIONS
2104  ;;;============================================================================
2105
2106  ;;;Gets segment-mates with a specified pos-2 from MORPH-COOCRECS-INDEXED.
2107  ;;;If threshold is non-nil, dynamically filters records to get mates over some threshold.
2108  ;;;If threshold is nil, collects all records for pos-2
2109  ;;;(this option would be used when the db has already been pre-filtered).
2110  ;;;If argument FILTER-SELF is t, a record showing that LEX-1 is its own segment-mate is
2111  ;;;filtered before return.
2112  (defun get-morph-window-mates-with-files
2113      (n pos-1 lex-1 pos-2 directory corpus-prefix
2114         &key (threshold nil)(threshold-key 'conditional-probability)(filter-self t)(db-pos-display 'english)
2115         (return-morphs-only nil))
2116    (let* ((output nil)
2117           (morph-coocrecs-indexed-file (format nil "~A~A.morph.coocrecs.indexed" directory corpus-prefix)))
2118      (if (null *morph-coocrecs-indexed*)
2119          (progn
2120            (format t "~%Loading morph-coocrecs-indexed from file ~a." morph-coocrecs-indexed-file)
2121            (load-morph-coocrecs-indexed morph-coocrecs-indexed-file)))
2122      (let* ((n-group (second (assoc n *morph-coocrecs-indexed*)))
2123             (pos-1-group (second (assoc pos-1 n-group)))
2124             (lex-1-group (second (assoc lex-1 pos-1-group)))
2125             (pos-2-group (second (assoc pos-2 lex-1-group)))
2126             (pos-2-group-morph-coocrecs
2127              (loop for partial-morph-coocrec in pos-2-group collect
2128                    (append (list n pos-1 lex-1 pos-2) partial-morph-coocrec)))
2129             (window-mates-including-self
2130              (if threshold
2131                  (loop for morph-coocrec in pos-2-group-morph-coocrecs when
2132                        (>= (get-morph-coocrec-element morph-coocrec threshold-key) threshold)
2133                        collect morph-coocrec)
2134                  pos-2-group-morph-coocrecs)))
2135        (if filter-self
2136            (let ((morph-coocrec-self
2137                   (make-and-get-morph-coocrec
2138                    n lex-1 pos-1 lex-1 pos-1 *morph-coocrecs-indexed*
2139                    :db-pos-display db-pos-display)))
2140
2141              (setq output (remove morph-coocrec-self window-mates-including-self)))
2142            (setq output window-mates-including-self))
2143        (if (and output return-morphs-only)
2144            (let ((temp-output nil))
2145              (loop for morph-coocrec in output do
2146                    (let* ((lex-2 (get-morph-coocrec-element morph-coocrec 'lex-2))
2147                           (pos-2 (get-morph-coocrec-element morph-coocrec 'pos-2)))
2148                      (setq temp-output
2149                            (adjoin (list lex-2 pos-2) temp-output :test #'equal))))
2150              (setq output temp-output)))
2151        output)))
2152
2153  ;;;<<Allow semantic smoothing when seeking window-mates for a morph.
2154  ;;;Look up cats for that morph, find the cats which they co-occur with,
2155  ;;;and the morphs which were associated with the co-occurring cats in the corpus.
2156  ;;;Iff LEVEL is specified, collect ancestors for all associated cats
2157  ;;;and then filter, using only cats at the right level.
```

LINE # TEXT

```
2158 ;;;Else any cat codes supplied by the thesaurus are used,
2159 ;;;and these can be at several levels.
2160 ;;;CATS keyword arg allows input of thesaurus codes;
2161 ;;;if cats are supplied, then we can skip trying to look up cats
2162 ;;;for this morph using GET-MORPH-CATS.
2163 (defun get-morph-window-mates-via-cats-with-files
2164     (n pos-1 lex-1 pos-2 directory corpus-prefix morphs-to-cats-prefix
2165        &key
2166        (level nil)
2167        (threshold nil)
2168        (threshold-key 'conditional-probability)
2169        (filter-self t)
2170        (thesaurus 'thesaurus-full)
2171        (return-morphs-only nil)
2172        (cats nil))
2173   (let* ((output nil)
2174          (morphs-to-cats-file
2175           (format nil "~A~A.morphs.to.cats"  directory morphs-to-cats-prefix)))
2176     (if (null *morphs-to-cats*)
2177         (progn
2178           (format t "~%Loading morph-to-cats from file ~a." morphs-to-cats-file)
2179           (load-morphs-to-cats morphs-to-cats-file)))
2180     (let ((cats-at-right-level nil)
2181           (cat-window-mates nil)
2182           (morphs-under-cat-window-mates nil))
2183
2184       ;;Iff LEVEL is specified, collect ancestors for all associated codes
2185       ;;and then filter, using only codes at the right level.
2186       ;;Else any codes supplied by the thesaurus are used,
2187       ;;and these can be at several levels.
2188       ;;If CATS arg is supplied, use its value, after
2189       ;;filtering to make sure supplied cats were at the specified semantic level.
2190       (if cats
2191           (loop for cat in cats do
2192                 (if (equal (length cat) level)
2193                     (setq cats-at-right-level
2194                           (adjoin cat cats-at-right-level :test #'equal))))
2195           ;;Else look up cats in *morphs-to-cats*
2196           (setq cats-at-right-level
2197                 (get-morph-cats lex-1 pos-1 *morphs-to-cats* :thesaurus thesaurus :level level)))
2198       (if cats-at-right-level
2199           (loop for cat-at-right-level in cats-at-right-level do
2200                 (loop for window-mate-for-cat-at-right-level in
2201                       (get-cat-window-mates-with-files
2202                        n cat-at-right-level directory corpus-prefix
2203                        :threshold threshold
2204                        :threshold-key threshold-key
2205                        :filter-self filter-self
2206                        :return-cats-only t)
2207                       do
2208                       (setq cat-window-mates
2209                             (adjoin window-mate-for-cat-at-right-level cat-window-mates :test #'equal)))))
2210       (if cat-window-mates
2211           (progn
2212             (loop for cat-window-mate in cat-window-mates do
2213                   (let ((cat-window-mate-morphs
2214                          (get-cat-morphs-with-files
2215                           cat-window-mate directory morphs-to-cats-prefix :pos pos-2)))
2216                     (loop for cat-window-mate-morph in cat-window-mate-morphs do
2217                           (setq morphs-under-cat-window-mates
2218                                 (adjoin cat-window-mate-morph morphs-under-cat-window-mates :test #'equal)))))
2219             (setq output morphs-under-cat-window-mates)))
2220
2221       ;;If we are to RETURN-MORPHS-ONLY.
2222       (if (and output return-morphs-only)
2223           (let ((temp-output nil))
2224             (loop for morphrec in output do
2225                   (let* ((lex (get-morphrec-element morphrec 'lex))
2226                          (pos (get-morphrec-element morphrec 'pos)))
2227                     (setq temp-output
2228                           (adjoin (list lex pos) temp-output :test #'equal))))
2229             (setq output temp-output)))
2230       output)))
2231
2232 ;;;<<Using the most specific info available,
2233 ;;;Seek window-mates for the specified morph.
2234 ;;;Try without semantic smoothing first.
2235 ;;;If we fail, try semantic smoothing at several levels,
2236 ;;;using the most specific level first.
2237 (defun get-morph-window-mates-most-specific-with-files
2238     (n pos-1 lex-1 pos-2 directory corpus-prefix morphs-to-cats-prefix
2239        &key (level nil)(threshold nil)(threshold-key 'conditional-probability)(filter-self t)
2240        (thesaurus 'thesaurus-full)
2241        (db-pos-display 'english)
2242        (return-morphs-only nil))
2243   (let* ((results nil))
2244     (let ((results-without-cats
2245            (get-morph-window-mates-with-files
2246             n pos-1 lex-1 pos-2 directory corpus-prefix
2247             :threshold threshold :threshold-key threshold-key
2248             :filter-self filter-self
2249             :db-pos-display db-pos-display
2250             :return-morphs-only return-morphs-only)))
2251
2252       (if results-without-cats
2253           (progn
2254             (format t "~%The following results were obtained for specific morphs in corpus, without semantic smoothing.")
2255             (setq results results-without-cats))
2256
2257           ;;If no results without cats, try using cats.
2258           ;;First determine whether any cats are listed for this morph in appropriate morphs-to-cats.
2259           (let ((cats
2260                  (get-morph-cats-with-files
2261                   lex-1 pos-1 directory morphs-to-cats-prefix
2262                   :thesaurus thesaurus
2263                   :level level)))
2264
2265             ;;If no cats are listed, the morph is either
2266             ;;(a) not in the current corpus or
2267             ;;(b) one of the corpus morphs with failed thesaurus lookup.
2268             ;;Try looking it up in the specified thesaurus.
2269             (if (null cats)
2270                 (progn
2271                   (format t "~%No cats listed for this morph in corpus. Trying lookup in ~s." thesaurus)
2272                   ;;Make dummy morphrec for use in thesaurus lookup.
2273                   (let* ((dummy-morphrec (make-morphrec lex-1 pos-1))
2274                          (thesaurus-entries
2275                           (get-thesaurus-entries-with-files
2276                            dummy-morphrec directory
2277                            :thesaurus thesaurus))
```

LINE # TEXT

```
                        (cats-from-thesaurus nil))

                    (loop for entry in thesaurus-entries do
                          (let ((cat-codes
                                 (get-thesaurus-entry-element entry 'cat-codes :thesaurus thesaurus)))
                            (loop for cat-code in cat-codes do
                                  (setq cats-from-thesaurus
                                        (adjoin cat-code cats-from-thesaurus :test #'equal)))))

                    ;;If there were no cats in the thesaurus
                    (if (null cats-from-thesaurus)
                        (format t "~%No cats listed for this morph in ~s." thesaurus)
                        ;;but if thesaurus search did yield cats use them.
                        (progn
                          (format t "~%Found cats ~s in ~s." cats-from-thesaurus thesaurus)
                          (setq cats cats-from-thesaurus)

                          ;;Seek window mates at the most specific levels first.
                          ;;Quit as soon as any result is found.
                          (loop for level from 4 downto 1 until results do
                                (setq results
                                      (get-morph-window-mates-via-cats-with-files
                                       n pos-1 lex-1 pos-2 directory corpus-prefix morphs-to-cats-prefix
                                       :level level
                                       :threshold threshold
                                       :threshold-key threshold-key
                                       :filter-self filter-self
                                       :thesaurus thesaurus
                                       :return-morphs-only return-morphs-only
                                       :cats cats))
                                (if results
                                    (format t "~%Semantic co-occurrence records at level ~s gave the following results." level))))))))))
    results))

;;;============================================================================
;;;  II.C.6.  MAKING MORPH-COOCRECS-SORTED
;;;============================================================================

;;;Makes MORPH-COOCRECS-SORTED for a corpus, using MORPH-COOCRECS-INDEXED as input.
(defun make-morph-coocrecs-sorted-with-files
    (n-range pos-1-list pos-2-list directory corpus-prefix
             &key (db-pos-display 'english)(threshold 0.1)(threshold-key 'conditional-probability))
  (let* ((morphrecs-indexed-file (format nil "~A~A.morphrecs.indexed" directory corpus-prefix))
         (morphrecs-indexed (read-from-file morphrecs-indexed-file))
         (morph-coocrecs-indexed-file (format nil "~A~A.morph.coocrecs.indexed" directory corpus-prefix))
         (morph-coocrecs-indexed (read-from-file morph-coocrecs-indexed-file))
         (morph-coocrecs-sorted
          (make-morph-coocrecs-sorted
           n-range pos-1-list pos-2-list morphrecs-indexed morph-coocrecs-indexed
           :db-pos-display db-pos-display :threshold threshold :threshold-key threshold-key))
         (morph-coocrecs-sorted-file (format nil "~A~A.morph.coocrecs.sorted" directory corpus-prefix)))
    (format t "Writing to ~S~%" morph-coocrecs-sorted-file)
    (write-to-file-no-pp morph-coocrecs-sorted morph-coocrecs-sorted-file)))

;;;Given MORPH-COOCRECS-INDEXED, collects all morph-coocrecs for specific pos's and n.
(defun make-morph-coocrecs-sorted
    (n-range pos-1-list pos-2-list morphrecs-indexed morph-coocrecs-indexed
             &key (threshold 0.1)(threshold-key 'conditional-probability)(db-pos-display 'english))
  (let (output)
    (loop for n from (first n-range) to (second n-range) do
          (loop for pos-1 in pos-1-list do
                (loop for lex-1 in (second (assoc pos-1 morphrecs-indexed)) do
                      (loop for pos-2 in pos-2-list do
                            (loop for lex-2 in
                                  (second (assoc pos-2 morphrecs-indexed)) do

                                  (let* ((morph-coocrec
                                          ;;<<CREATION OF MORPH-COOCREC.
                                          ;;<<CONVERTS PARTIAL MORPH-COOCREC TO COMPLETE ONE AT FETCH TIME.
                                          (make-and-get-morph-coocrec
                                           n lex-1 pos-1 lex-2 pos-2 morph-coocrecs-indexed
                                           :db-pos-display db-pos-display))
                                         (threshold-key-value
                                          (get-morph-coocrec-element
                                           morph-coocrec threshold-key)))
                                    (if (>= threshold-key-value threshold)
                                        (setq output (cons morph-coocrec output)))))))))
    ;;<<Sorting by mutual-information-smoothed -- fixed key.
    (sort-morph-coocrecs output threshold-key)))

;;;============================================================================
;;;  II.C.7.  SORTING MORPH-COOCRECS-SORTED
;;;============================================================================

(defun sort-morph-coocrecs (morph-coocrecs-sorted element-name)
  (let ((position (get-morph-coocrec-element-position element-name)))
    ;;LAMBDA means "test whether list-1 is greater than list-2
    ;;with respect to the elements at nth position"
    (sort morph-coocrecs-sorted
          #'(lambda (list1 list2)
              (if (not (numberp (nth position list1)))
                  (error "can't sort: pointer to non-numerical element in morph-coocrec")
                  (> (nth position list1)(nth position list2)))))))

;;;============================================================================
;;;  II.C.8.  QUERYING MORPH-COOCRECS-SORTED
;;;============================================================================

;;;Collects list of e.g. conditional-probability values over all MORPH-COOCRECS-SORTED
(defun get-all-morph-coocrec-elements
    (element-name morph-coocrecs-sorted)
  (loop for morph-coocrec in morph-coocrecs-sorted collect
        (get-morph-coocrec-element morph-coocrec element-name)))

;;;============================================================================
;;;  II.C.9.  DEFINING MORPH-COOCRECS-DIFFREC
;;;============================================================================

;;;A MORPH-COOCREC-DIFFREC is a record of the comparison between
;;;comparable MORPH-COOCRECs in two corpora.
(defvar *morph-coocrec-diffrec-elements*
  '((n 0)
    (lex-1 1)
    (pos-1 2)
    (lex-2 3)
    (pos-2 4)
    (element-1 5)
    (element-2 6)
    (difference 7)
    (count-1 8) ;morph-1 segment count in corpus-1
```

LINE # TEXT

```
2398      (count-2 9) ;morph-1 segment count in corpus-1
2399      (count-sum 10))) ;morph-1 segment count in corpus-1
2400
2401  (defun make-morph-coocrec-diffrec (n lex-1 pos-1 lex-2 pos-2 element-1 element-2 difference count-1 count-2 count-sum)
2402    (let ((morph-coocrec-diffrec
2403           (loop for element in *morph-coocrec-diffrec-elements* collect nil)))
2404      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'n n))
2405      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'lex-1 lex-1))
2406      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'pos-1 pos-1))
2407      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'lex-2 lex-2))
2408      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'pos-2 pos-2))
2409      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'element-1 element-1))
2410      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'element-2 element-2))
2411      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'difference difference))
2412      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'count-1 count-1))
2413      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'count-2 count-2))
2414      (setq morph-coocrec-diffrec (put-morph-coocrec-diffrec-element morph-coocrec-diffrec 'count-sum count-sum))
2415      morph-coocrec-diffrec))
2416
2417  (defun get-morph-coocrec-diffrec-element-position (element-name)
2418    (second
2419     (assoc element-name *morph-coocrec-diffrec-elements*)))
2420
2421  (defun get-morph-coocrec-diffrec-element (morph-coocrec-diffrec element-name)
2422    (let ((position (get-morph-coocrec-diffrec-element-position element-name)))
2423      (nth position morph-coocrec-diffrec)))
2424
2425  (defun put-morph-coocrec-diffrec-element (morph-coocrec-diffrec element-name value)
2426    (let ((morph-coocrec-diffrec-copy (copy-list morph-coocrec-diffrec))
2427          (position (get-morph-coocrec-diffrec-element-position element-name)))
2428      (setf (nth position morph-coocrec-diffrec-copy) value)
2429      morph-coocrec-diffrec-copy))
2430
2431  (defun add-morph-coocrec-diffrec-element (morph-coocrec-diffrec element-name value)
2432    (let ((morph-coocrec-diffrec-copy (copy-list morph-coocrec-diffrec))
2433          (old-value (get-morph-coocrec-diffrec-element morph-coocrec-diffrec element-name))
2434          (new-value nil)
2435          (position (get-morph-coocrec-diffrec-element-position element-name)))
2436      (if (not (listp old-value))
2437          (error "old value not a list")
2438          (progn (setq new-value (adjoin value old-value :test #'equal))
2439                 (setf (nth position morph-coocrec-diffrec-copy) new-value)))
2440      morph-coocrec-diffrec-copy))
2441
2442  (defun sort-morph-coocrec-diffrecs (morph-coocrec-diffrecs element-name)
2443    (let ((position (get-morph-coocrec-diffrec-element-position element-name)))
2444      ;;LAMBDA means "test whether list-1 is greater than list-2
2445      ;;with respect to the elements at nth position"
2446      (sort morph-coocrec-diffrecs
2447            #'(lambda (list1 list2)
2448                (if (not (numberp (nth position list1)))
2449                    (error "can't sort: pointer to non-numerical element in rec")
2450                    (> (nth position list1)(nth position list2)))))))
2451
2452  ;;;============================================================================
2453  ;;;  II.C.10.  COMPARING MORPH-COOCRECS-SORTED
2454  ;;;============================================================================
2455
2456  ;;;Compares MORPH-COOCRECS-SORTED in two corpora.
2457  ;;;Assumes MORPH-COOCRECS-SORTED-FILES have been presorted as desired.
2458  (defun compare-morph-coocrecs-sorted
2459      (directory corpus-prefix-1 corpus-prefix-2
2460                 &key (element 'conditional-probability))
2461
2462    (let* ((morph-coocrecs-sorted-file-1 (format nil "~A~A.morph.coocrecs.sorted"  directory corpus-prefix-1))
2463           (morph-coocrecs-indexed-file-2 (format nil "~A~A.morph.coocrecs.indexed"  directory corpus-prefix-2))
2464           (morph-coocrecs-sorted-file-2 (format nil "~A~A.morph.coocrecs.sorted"  directory corpus-prefix-2))
2465           (output-file (format nil "~A~A.vs.~A.morph.coocrecs.comparison"  directory corpus-prefix-1 corpus-prefix-2))
2466           (morph-coocrecs-1 (read-from-file morph-coocrecs-sorted-file-1))
2467           (pairs-1
2468            (loop for record in morph-coocrecs-1 collect
2469                  (list
2470                   (get-morph-coocrec-element record 'n)
2471                   (get-morph-coocrec-element record 'lex-1)
2472                   (get-morph-coocrec-element record 'pos-1)
2473                   (get-morph-coocrec-element record 'lex-2)
2474                   (get-morph-coocrec-element record 'pos-2))))
2475           (morph-coocrecs-indexed-2 (read-from-file morph-coocrecs-indexed-file-2))
2476           (morph-coocrecs-2 (read-from-file morph-coocrecs-sorted-file-2))
2477           (pairs-2
2478            (loop for record in morph-coocrecs-2 collect
2479                  (list
2480                   (get-morph-coocrec-element record 'n)
2481                   (get-morph-coocrec-element record 'lex-1)
2482                   (get-morph-coocrec-element record 'pos-1)
2483                   (get-morph-coocrec-element record 'lex-2)
2484                   (get-morph-coocrec-element record 'pos-2))))
2485
2486           (in-both (intersection pairs-1 pairs-2 :test #'equal))
2487           (in-both-count (length in-both))
2488           (in-pairs-1-but-not-in-pairs-2 (set-difference pairs-1 pairs-2 :test  #'equal))
2489           (in-pairs-2-but-not-in-pairs-1 (set-difference pairs-2 pairs-1 :test  #'equal))
2490           (in-both-difference-records-corpus-1-order nil)
2491           (in-both-difference-records-least-difference-order nil)
2492           (in-both-difference-records-greatest-count-sum-order nil)
2493           (summed-in-both-differences 0.0)
2494           (average-in-both-differences nil)
2495           (average-pairs-1-element nil)
2496           (average-pairs-2-element nil)
2497           (summed-pairs-1-elements 0.0)
2498           (summed-pairs-2-elements 0.0)
2499
2500           (morphrecs-indexed-file-1 (format nil "~A~A.morphrecs.indexed"  directory corpus-prefix-1))
2501           (morphrecs-indexed-1 (read-from-file morphrecs-indexed-file-1))
2502           (morphrecs-indexed-file-2 (format nil "~A~A.morphrecs.indexed"  directory corpus-prefix-2))
2503           (morphrecs-indexed-2 (read-from-file morphrecs-indexed-file-2)))
2504
2505      (if output-file
2506          (if (probe-file output-file)
2507              (delete-file output-file)))
2508
2509      ;;Loop through MORPH-COOCRECS-SORTED-FILE-1, the training corpus,
2510      ;;so that this ordering will be preserved.
2511      ;;If retrieved info shows a pair which is in both corpora, go ahead.
2512      (loop for record-1 in morph-coocrecs-1 do
2513            (let ((n (get-morph-coocrec-element record-1 'n))
2514                  (lex-1 (get-morph-coocrec-element record-1 'lex-1))
2515                  (pos-1 (get-morph-coocrec-element record-1 'pos-1))
2516                  (lex-2 (get-morph-coocrec-element record-1 'lex-2))
2517                  (pos-2 (get-morph-coocrec-element record-1 'pos-2)))
```

LINE # TEXT

```
2518            (if (member (list n lex-1 pos-1 lex-2 pos-2) in-both :test #'equal)
2519                (let* ((record-2
2520                        ;;<<RE-CREATION OF MORPH-COOCREC at fetch time.
2521                        (make-and-get-morph-coocrec
2522                         n lex-1 pos-1 lex-2 pos-2 morph-coocrecs-indexed-2))
2523                       ;;ELEMENT is e.g. CONDITIONAL-PROBABILITY.
2524                       (element-1 (get-morph-coocrec-element record-1 element))
2525                       (element-2 (get-morph-coocrec-element record-2 element))
2526                       (difference
2527                        (round-to-n-digits
2528                         (abs (- element-1 element-2)) *significant-digits*)))
2529
2530                  ;;MAKE A NEW DIFFERENCE RECORD.
2531                  (let* ((morphrec-1 (first (get-morphrecs lex-1 morphrecs-indexed-1 :pos pos-1)))
2532                         ;;<<Get same info -- for LEX-1 and POS-1 -- from MORPHREC-2
2533                         (morphrec-2 (first (get-morphrecs lex-1 morphrecs-indexed-2 :pos pos-1)))
2534                         (count-1 (get-morphrec-element morphrec-1 'morph-segments-count))
2535                         (count-2 (get-morphrec-element morphrec-2 'morph-segments-count))
2536                         (count-sum (+ count-1 count-2))
2537                         (new-morph-coocrec-diffrec
2538                          (make-morph-coocrec-diffrec
2539                           n lex-1 pos-1 lex-2 pos-2 element-1 element-2 difference count-1 count-2 count-sum)))
2540
2541                    (setq in-both-difference-records-corpus-1-order
2542                          (cons
2543                           new-morph-coocrec-diffrec
2544                           in-both-difference-records-corpus-1-order)))
2545
2546                  ;;Maintain running sums.
2547                  (setq summed-in-both-differences (+ summed-in-both-differences difference))
2548                  (setq summed-pairs-1-elements (+ summed-pairs-1-elements element-1))
2549                  (setq summed-pairs-2-elements (+ summed-pairs-2-elements element-2))))))
2550
2551      (setq in-both-difference-records-corpus-1-order (reverse in-both-difference-records-corpus-1-order))
2552      (setq in-both-difference-records-least-difference-order
2553            (reverse (sort-morph-coocrec-diffrecs in-both-difference-records-corpus-1-order 'difference))) ;<<reverse
2554      (setq in-both-difference-records-greatest-count-sum-order
2555            (sort-morph-coocrec-diffrecs in-both-difference-records-corpus-1-order 'count-sum))
2556
2557      ;;Get averages if there are any shared records.
2558      ;;If not, averages remain nil.
2559      (if in-both-count
2560          (progn
2561            (setq average-in-both-differences
2562                  (round-to-n-digits
2563                   (/ (float summed-in-both-differences) (float in-both-count)) *significant-digits*))
2564            (setq average-pairs-1-element
2565                  (round-to-n-digits
2566                   (/ (float summed-pairs-1-elements) (float in-both-count)) *significant-digits*))
2567            (setq average-pairs-2-element
2568                  (round-to-n-digits
2569                   (/ (float summed-pairs-2-elements) (float in-both-count)) *significant-digits*))))
2570
2571      ;;PRINTING STARTS
2572      (smart-format output-file "~%;;;~A and ~A ~%;;;compared in terms of ~s"
2573                    morph-coocrecs-sorted-file-1 morph-coocrecs-sorted-file-2 element)
2574      (smart-format output-file "~2%(average-~s-corpus-1 ~s)" element average-pairs-1-element)
2575      (smart-format output-file "~2%(average-~s-corpus-2 ~s)" element average-pairs-2-element)
2576      (smart-format output-file "~2%(average-~s-difference ~s)"
2577                    element average-in-both-differences)
2578
2579      (smart-format output-file "~2%(in-both-pairs-count ~s)"
2580                    (length in-both-difference-records-corpus-1-order))
2581      (smart-format output-file "~2%(difference-records-corpus-1-order ~s)"
2582                    in-both-difference-records-corpus-1-order)
2583
2584      (smart-format output-file "~2%(difference-records-least-difference-order ~s)"
2585                    in-both-difference-records-least-difference-order)
2586
2587      (smart-format output-file "~2%(difference-records-greatest-count-sum-order ~s)"
2588                    in-both-difference-records-greatest-count-sum-order)
2589
2590      (smart-format output-file "~2%(in-corpus-1-but-not-corpus-2-count ~s)"
2591                    (length in-pairs-1-but-not-in-pairs-2))
2592      (smart-format output-file "~2%(in-corpus-1-but-not-corpus-2 ~s)" in-pairs-1-but-not-in-pairs-2)
2593
2594      (smart-format output-file "~2%(in-corpus-2-but-not-corpus-1-count ~s)"
2595                    (length in-pairs-2-but-not-in-pairs-1))
2596      (smart-format output-file "~2%(in-corpus-2-but-not-corpus-1 ~s)" in-pairs-2-but-not-in-pairs-1)))
2597
2598  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2599  ;;;  II.D. DEFINING NODE-STRUCTURE
2600  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2601
2602  (defvar *node-structure-elements*
2603    '((object 0)
2604      (daughters 1)
2605      (catrec 2)
2606      (morphrecs 3)))
2607
2608  (defun make-node-structure (object)
2609    (let ((node-structure
2610           (loop for element in *node-structure-elements* collect nil)))
2611      (setq node-structure (put-node-structure-element node-structure 'object object))
2612      node-structure))
2613
2614  (defun get-node-structure-element-position (element-name)
2615    (second
2616     (assoc element-name
2617            *node-structure-elements*)))
2618
2619  (defun get-node-structure-element (node-structure element-name)
2620    (let ((position (get-node-structure-element-position element-name)))
2621      (nth position node-structure)))
2622
2623  ;;;Non-destructive: returns a copy of the input node-structure with
2624  ;;;the specified modification.
2625  (defun put-node-structure-element (node-structure element-name value)
2626    (let ((node-structure-copy (copy-list node-structure))
2627          (position (get-node-structure-element-position element-name)))
2628      (setf (nth position node-structure-copy) value)
2629      node-structure-copy))
2630
2631  (defun add-node-structure-element (node-structure element-name value)
2632    (let ((node-structure-copy (copy-list node-structure))
2633          (old-value (get-node-structure-element node-structure element-name))
2634          (new-value nil)
2635          (position (get-node-structure-element-position element-name)))
2636      (if (not (listp old-value))
2637          (error "old value not a list")
```

LINE # | TEXT

```
        (progn (setq new-value (adjoin value old-value :test #'equal))
               (setf (nth position node-structure-copy) new-value)))
    node-structure-copy))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  II.E. CATRECS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;==============================================================================
;;;  II.E.1.  DEFINING CATREC DATASTRUCTURE
;;;==============================================================================

(defvar *catrec-elements*
  '((code 0)
    (segments 1)
    (hit-count 2)
    (cat-segments-count 3)
    (occur-in-segment-probability 4)
    (occur-in-segment-information 5)
    (occur-in-segment-probability-smoothed 6)
    (cat-unigram-probability 7)
    (cat-unigram-information 8)
    (cat-unigram-probability-smoothed 9)))

(defun make-catrec (code)
  (let ((catrec
         (loop for element in *catrec-elements* collect nil)))
    (setq catrec (put-catrec-element catrec 'code code))
    catrec))

(defun get-catrec-element-position (element-name)
  (second
   (assoc element-name
          *catrec-elements*)))

(defun get-catrec-element (catrec element-name)
  (let ((position (get-catrec-element-position element-name)))
    (nth position catrec)))

(defun put-catrec-element (catrec element-name value)
  (let ((catrec-copy (copy-list catrec))
        (position (get-catrec-element-position element-name)))
    (setf (nth position catrec-copy) value)
    catrec-copy))

(defun add-catrec-element (catrec element-name value)
  (let ((catrec-copy (copy-list catrec))
        (old-value (get-catrec-element catrec element-name))
        (new-value nil)
        (position (get-catrec-element-position element-name)))
    (if (not (listp old-value))
        (error "Old value not a list.")
        (progn (setq new-value (adjoin value old-value :test #'equal))
               (setf (nth position catrec-copy) new-value)))
    catrec-copy))

;;;==============================================================================
;;;  II.E.2.  MAKING CATRECS.SORTED
;;;==============================================================================

(defun make-catrecs-sorted-with-files
    (directory corpus-prefix morphs-to-cats-prefix
               &key (thesaurus 'thesaurus-full))
  (let* ((morphrecs-sorted-file (format nil "~A~A.morphrecs.sorted" directory corpus-prefix))
         (morphs-to-cats-file (format nil "~A~A.morphs.to.cats"  directory morphs-to-cats-prefix))
         (morphs-global-file (format nil "~A~A.morphs.global" directory corpus-prefix))
         (catrecs-sorted-file (format nil "~A~A.catrecs.sorted" directory corpus-prefix))
         (morphrecs-sorted (read-from-file morphrecs-sorted-file))
         (morphs-to-cats (read-from-file morphs-to-cats-file))
         (catrecs-sorted
          (make-catrecs-sorted morphrecs-sorted morphs-to-cats morphs-global-file :thesaurus thesaurus)))
    (write-to-file-no-pp catrecs-sorted catrecs-sorted-file)))

(defun make-catrecs-sorted (morphrecs-sorted morphs-to-cats morphs-global-file &key (thesaurus 'thesaurus-full))
  (let* ((prelim-results (make-catrecs-sorted-preliminary morphrecs-sorted morphs-to-cats :thesaurus thesaurus))
         (total-number-of-cats (first prelim-results))
         (number-of-different-cats (second prelim-results))
         (prelim-catrecs-sorted (third prelim-results))
         (augmented-catrecs-sorted
          (augment-catrecs-sorted
           prelim-catrecs-sorted
           total-number-of-cats
           number-of-different-cats
           morphs-global-file)))
    augmented-catrecs-sorted))

(defun make-catrecs-sorted-preliminary (morphrecs-sorted morphs-to-cats &key (thesaurus 'thesaurus-full))
  (let ((catrecs-sorted nil)
        (total-number-of-cats 0)
        (number-of-different-cats 0))
    (loop for morphrec in morphrecs-sorted do
          (let* ((lex (get-morphrec-element morphrec 'lex))
                 (pos (get-morphrec-element morphrec 'pos))
                 (morph-segments (get-morphrec-element morphrec 'segments))
                 (morph-hit-count (get-morphrec-element morphrec 'hit-count))
                 (terminal-cats (get-morph-cats lex pos morphs-to-cats :thesaurus thesaurus))
                 ;;Initialize here, add ancestors below.
                 (cats-for-this-morph terminal-cats))

            ;;Include ancestor-cats in cats-for-this-morph.
            (let ((ancestor-cats nil))
              (loop for terminal-cat in terminal-cats do
                    (loop for ancestor-cat in (reverse (get-code-ancestors terminal-cat)) do
                          (setq ancestor-cats
                                (adjoin ancestor-cat ancestor-cats :test #'equal))))
              (loop for ancestor-cat in ancestor-cats do
                    (setq cats-for-this-morph
                          (adjoin ancestor-cat cats-for-this-morph :test #'equal))))

            ;;For each cat, including ancestors, UPDATE OR MAKE A CATREC.
            (loop for cat-for-this-morph in cats-for-this-morph do

                  ;;Update TOTAL-NUMBER-OF-CATS.
                  (setq total-number-of-cats
                        (+ morph-hit-count total-number-of-cats))

                  (let ((existing-catrec
                         (assoc cat-for-this-morph catrecs-sorted :test #'equal)))
                    (if existing-catrec
                        ;;Add segments in which CAT-FOR-THIS-MORPH appears to
```

LINE # TEXT

```
2758                        ;;segments in which abstract-level cat appears if it is not
2759                        ;;already there.
2760                        (let ((existing-catrec-segments
2761                               (get-catrec-element existing-catrec 'segments))
2762                              (existing-catrec-hit-count
2763                               (get-catrec-element existing-catrec 'hit-count))
2764                              (new-catrec existing-catrec))
2765                          (loop for morph-segment in morph-segments do
2766                                (setq existing-catrec-segments
2767                                      (adjoin morph-segment existing-catrec-segments :test #'equal)))
2768                          (setq existing-catrec-segments
2769                                (sort existing-catrec-segments #'>))
2770                          (setq existing-catrec-hit-count
2771                                (+ morph-hit-count existing-catrec-hit-count))
2772                          (setq new-catrec
2773                                (put-catrec-element new-catrec 'segments existing-catrec-segments))
2774                          (setq new-catrec
2775                                (put-catrec-element new-catrec 'hit-count existing-catrec-hit-count))
2776                          (setq catrecs-sorted
2777                                (subst new-catrec
2778                                       existing-catrec
2779                                       catrecs-sorted)))
2780                        ;;Else there was no EXISTING-CATREC,
2781                        ;;so make a new catrec.
2782                        (progn
2783                          ;;Update NUMBER-OF-DIFFERENT-CATS.
2784                          (setq  number-of-different-cats
2785                                 (1+ number-of-different-cats))
2786
2787                          ;;<<CREATION OF CATREC
2788                          (let* ((new-catrec (make-catrec cat-for-this-morph)))
2789                            (setq morph-segments (sort morph-segments #'>))
2790                            (setq new-catrec (put-catrec-element new-catrec 'segments morph-segments))
2791                            (setq new-catrec (put-catrec-element new-catrec 'hit-count morph-hit-count))
2792                            (setq catrecs-sorted (cons new-catrec catrecs-sorted)))))))))
2793      (list
2794       total-number-of-cats
2795       number-of-different-cats
2796       (reverse catrecs-sorted))))
2797
2798  ;;;Takes preliminary CATRECS-SORTED.
2799  ;;;Visits each catrec, calculating and adding additional material.
2800  ;;;Returns augmented and completed CATRECS-SORTED.
2801  (defun augment-catrecs-sorted
2802      (old-catrecs total-number-of-cats number-of-different-cats morphs-global-file)
2803
2804    (let ((all-segments-count
2805           (get-global-value-from-file  morphs-global-file 'all-segments-count)))
2806
2807      (loop for old-catrec in old-catrecs collect
2808            (let* ((new-catrec old-catrec)
2809                   ;;SEGMENTS THIS CAT APPEARS IN.
2810                   (cat-segments
2811                    (get-catrec-element old-catrec 'segments))
2812                   ;;COUNT OF SEGMENTS THIS CAT APPEARS IN.
2813                   (cat-segments-count (length cat-segments))
2814                   ;;OF ALL SEGMENTS IN CORPUS, WHAT PROPORTION CONTAIN THIS CAT?
2815                   (occur-in-segment-probability
2816                    (round-to-n-digits
2817                     (/ (float cat-segments-count)(float all-segments-count))
2818                     *significant-digits*))
2819                   (occur-in-segment-information
2820                    (round-to-n-digits
2821                     ;;<<log base 2
2822                     (log (/ (float 1) (float occur-in-segment-probability)) 2)
2823                     *significant-digits*))
2824                   ;;USING 1+ PROBABILITY SMOOTHING.
2825                   (occur-in-segment-probability-smoothed
2826                    (round-to-n-digits
2827                     (/ (+ (float cat-segments-count)(float 1))
2828                        (+ (float 2)(float all-segments-count)))
2829                     *significant-digits*))
2830                   (hits-this-cat (get-catrec-element old-catrec 'hit-count))
2831                   (cat-unigram-probability
2832                    (round-to-n-digits
2833                     (/ (float hits-this-cat)(float total-number-of-cats))
2834                     *significant-digits*))
2835                   (cat-unigram-information
2836                    (round-to-n-digits
2837                     ;;<<log base 2
2838                     (log (/ (float 1) (float cat-unigram-probability)) 2)
2839                     *significant-digits*))
2840                   (cat-unigram-probability-smoothed
2841                    (round-to-n-digits
2842                     (/ (+ (float hits-this-cat)(float  1))
2843                        (+ (float total-number-of-cats)
2844                           (float number-of-different-cats)))
2845                     *significant-digits*)))
2846
2847              ;;<<AUGMENTATION OF CATREC
2848              (setq new-catrec
2849                    (put-catrec-element new-catrec 'cat-segments-count cat-segments-count))
2850              (setq new-catrec
2851                    (put-catrec-element new-catrec 'occur-in-segment-probability occur-in-segment-probability))
2852              (setq new-catrec
2853                    (put-catrec-element new-catrec 'occur-in-segment-information occur-in-segment-information))
2854              (setq new-catrec
2855                    (put-catrec-element new-catrec 'occur-in-segment-probability-smoothed occur-in-segment-probability-smoothed))
2856              (setq new-catrec
2857                    (put-catrec-element new-catrec 'cat-unigram-probability cat-unigram-probability))
2858              (setq new-catrec
2859                    (put-catrec-element new-catrec 'cat-unigram-information cat-unigram-information))
2860              (setq new-catrec
2861                    (put-catrec-element new-catrec 'cat-unigram-probability-smoothed cat-unigram-probability-smoothed))))))
2862
2863  ;;;============================================================================
2864  ;;;  II.E.3.  QUERYING CATRECS-SORTED
2865  ;;;============================================================================
2866
2867  (defvar *CATRECS-SORTED* nil)
2868
2869  (defun load-catrecs-sorted
2870      (catrecs-sorted-file)
2871    (setq *CATRECS-SORTED*
2872          (read-from-file catrecs-sorted-file))
2873    (format t "~%LOADING COMPLETED"))
2874
2875  ;;;Collects list of e.g. CAT-UNIGRAM-PROBABILITY values over all cat records.
2876  (defun get-all-catrec-elements
2877      (element-name catrecs-sorted)
```

```
2878    (loop for catrec in catrecs-sorted collect
2879          (get-catrec-element catrec element-name)))
2880
2881  ;;;Cat records contain pre-computed pairs like CAT-UNIGRAM-PROBABILITY and CAT-UNIGRAM-INFORMATION.
2882  ;;;This function gets the SUM OF THE PRODUCTS of these pairs OVER ALL CATS.
2883  (defun get-catrec-element-entropy (probability-element information-element catrecs-sorted)
2884    (let* ((probabilities (get-all-catrec-elements probability-element catrecs-sorted))
2885           (informations (get-all-catrec-elements information-element catrecs-sorted))
2886           (products (loop for probability in probabilities and information in informations collect
2887                           (* probability information)))
2888           (sum 0.0))
2889      (loop for product in products do
2890            (setq sum (+ sum product)))
2891      sum))
2892
2893  ;;;============================================================================
2894  ;;;  II.E.4.  DEFINING CATREC-DIFFREC
2895  ;;;============================================================================
2896
2897  ;;;A catrec-diffrec is a record of the difference between corresponding catrecs in
2898  ;;;two corpora.
2899  (defvar *catrec-diffrec-elements*
2900    '((code 0)
2901      (element-1 1)
2902      (element-2 2)
2903      (difference 3)
2904      (count-1 4) ;cat-1 hit-count in corpus 1
2905      (count-2 5) ;cat-1 hit-count in corpus 2
2906      (count-sum 6)))
2907
2908  (defun make-catrec-diffrec (code element-1 element-2 difference count-1 count-2 count-sum)
2909    (let ((catrec-diffrec
2910           (loop for element in *catrec-diffrec-elements* collect nil)))
2911      (setq catrec-diffrec (put-catrec-diffrec-element catrec-diffrec 'code code))
2912      (setq catrec-diffrec (put-catrec-diffrec-element catrec-diffrec 'element-1 element-1))
2913      (setq catrec-diffrec (put-catrec-diffrec-element catrec-diffrec 'element-2 element-2))
2914      (setq catrec-diffrec (put-catrec-diffrec-element catrec-diffrec 'difference difference))
2915      (setq catrec-diffrec (put-catrec-diffrec-element catrec-diffrec 'count-1 count-1))
2916      (setq catrec-diffrec (put-catrec-diffrec-element catrec-diffrec 'count-2 count-2))
2917      (setq catrec-diffrec (put-catrec-diffrec-element catrec-diffrec 'count-sum count-sum))
2918      catrec-diffrec))
2919
2920  (defun get-catrec-diffrec-element-position (element-name)
2921    (second
2922     (assoc element-name *catrec-diffrec-elements*)))
2923
2924  (defun get-catrec-diffrec-element (catrec-diffrec element-name)
2925    (let ((position (get-catrec-diffrec-element-position element-name)))
2926      (nth position catrec-diffrec)))
2927
2928  (defun put-catrec-diffrec-element (catrec-diffrec element-name value)
2929    (let ((catrec-diffrec-copy (copy-list catrec-diffrec))
2930          (position (get-catrec-diffrec-element-position element-name)))
2931      (setf (nth position catrec-diffrec-copy) value)
2932      catrec-diffrec-copy))
2933
2934  (defun add-catrec-diffrec-element (catrec-diffrec element-name value)
2935    (let ((catrec-diffrec-copy (copy-list catrec-diffrec))
2936          (old-value (get-catrec-diffrec-element catrec-diffrec element-name))
2937          (new-value nil)
2938          (position (get-catrec-diffrec-element-position element-name)))
2939      (if (not (listp old-value))
2940          (error "old value not a list")
2941          (progn (setq new-value (adjoin value old-value :test #'equal))
2942                 (setf (nth position catrec-diffrec-copy) new-value)))
2943      catrec-diffrec-copy))
2944
2945  (defun sort-catrec-diffrecs (catrec-diffrecs element-name)
2946    (let ((position (get-catrec-diffrec-element-position element-name)))
2947      ;;LAMBDA means "test whether list-1 is greater than list-2
2948      ;;with respect to the elements at nth position"
2949      (sort catrec-diffrecs
2950            #'(lambda (list1 list2)
2951                (if (not (numberp (nth position list1)))
2952                    (error "can't sort: pointer to non-numerical element in rec")
2953                    (> (nth position list1)(nth position list2)))))))
2954
2955  ;;;============================================================================
2956  ;;;  II.E.5.  COMPARING CATRECS-SORTED
2957  ;;;============================================================================
2958
2959  ;;;Compares catrecs-sorted in two corpora.
2960  ;;;Assumes CATRECS-SORTED-FILES have been presorted as desired.
2961  (defun compare-catrecs-sorted
2962      (directory corpus-prefix-1 corpus-prefix-2
2963                 &key (element 'cat-unigram-probability))
2964
2965    (let* ((catrecs-indexed-file-1 (format nil "~A~A.catrecs.indexed"  directory corpus-prefix-1))
2966           (catrecs-sorted-file-1 (format nil "~A~A.catrecs.sorted"  directory corpus-prefix-1))
2967           (catrecs-indexed-file-2 (format nil "~A~A.catrecs.indexed"  directory corpus-prefix-2))
2968           (catrecs-sorted-file-2 (format nil "~A~A.catrecs.sorted"  directory corpus-prefix-2))
2969           (output-file (format nil "~A~A.vs.~A.catrecs.comparison"  directory corpus-prefix-1 corpus-prefix-2))
2970           (catrecs-indexed-1 (read-from-file catrecs-indexed-file-1))
2971           (catrecs-indexed-2 (read-from-file catrecs-indexed-file-2))
2972           (catrecs-sorted-1 (read-from-file catrecs-sorted-file-1))
2973           (cats-1
2974            (loop for cat in catrecs-sorted-1 collect
2975                  (get-catrec-element cat 'code)))
2976           (catrecs-sorted-2 (read-from-file catrecs-sorted-file-2))
2977           (cats-2
2978            (loop for cat in catrecs-sorted-2 collect
2979                  (get-catrec-element cat 'code)))
2980           (in-both (intersection cats-1 cats-2 :test #'equal))
2981           (in-both-count (length in-both))
2982           (in-cats-1-but-not-in-cats-2 (set-difference cats-1 cats-2 :test  #'equal))
2983           (in-cats-2-but-not-in-cats-1 (set-difference cats-2 cats-1 :test  #'equal))
2984           (in-both-difference-records nil)
2985           (summed-in-both-differences 0.0)
2986           (average-in-both-differences nil)
2987           (average-cats-1-element nil)
2988           (average-cats-2-element nil)
2989           (summed-cats-1-elements 0.0)
2990           (summed-cats-2-elements 0.0)
2991           (relative-entropy 0.0))
2992
2993      (if output-file
2994          (if (probe-file output-file)
2995              (delete-file output-file)))
2996
2997      ;;Loop through CATRECS-SORTED-1, the training corpus,
```

```
2998      ;;so that this ordering will be preserved.
2999      ;;If RECORD-1 is for a cat which is in both corpora, go ahead.
3000      (loop for record-1 in catrecs-sorted-1 do
3001            (let ((code (get-catrec-element record-1 'code)))
3002              (if (member code in-both :test #'equal)
3003                  (let* ((record-2 (get-catrec code catrecs-indexed-2))
3004                         ;;element is e.g. morph-unigram-probability
3005                         (element-1 (get-catrec-element record-1 element))
3006                         (element-2 (get-catrec-element record-2 element))
3007                         (difference
3008                          (round-to-n-digits
3009                           (abs (- element-1 element-2)) *significant-digits*)))
3010
3011                    ;;Make a new DIFFERENCE-RECORD, showing the respective values
3012                    ;;for the factor of interest in the two corpora
3013                    ;;and the difference between them.
3014                    (let* ((catrec-1 (get-catrec code catrecs-indexed-1))
3015                           (catrec-2 (get-catrec code catrecs-indexed-2))
3016                           (count-1 (get-catrec-element catrec-1 'cat-segments-count))
3017                           (count-2 (get-catrec-element catrec-2 'cat-segments-count))
3018                           (count-sum (+ count-1 count-2))
3019                           (new-catrec-diffrec
3020                            (make-catrec-diffrec
3021                             code element-1 element-2 difference count-1 count-2 count-sum)))
3022
3023                      (setq in-both-difference-records
3024                            (cons
3025                             new-catrec-diffrec
3026                             in-both-difference-records)))
3027
3028                    ;;Maintain running sums.
3029                    (setq summed-in-both-differences (+ summed-in-both-differences difference))
3030                    (setq summed-cats-1-elements (+ summed-cats-1-elements element-1))
3031                    (setq summed-cats-2-elements (+ summed-cats-2-elements element-2))
3032
3033                    ;;<<Kuhllback-Leibler formula for "relative entropy"
3034                    ;;Pi log 1/Qi
3035                    (if (equal element 'cat-unigram-probability)
3036                        ;;Pi = element-2 (test corpus), Qi = element-1 (training corpus)
3037                        (let ((summed (* element-2 (log (/ (float element-2)(float element-1)) 2))))
3038                          (setq relative-entropy (+ relative-entropy summed))))))))
3039
3040      (setq in-both-difference-records (reverse in-both-difference-records))
3041
3042      (setq relative-entropy (round-to-n-digits relative-entropy *significant-digits*))
3043
3044      ;;Get averages if any shared records.
3045      ;;If not, averages remain nil.
3046      (if in-both-count
3047          (progn
3048            (setq average-in-both-differences
3049                  (round-to-n-digits
3050                   (/ (float summed-in-both-differences) (float in-both-count)) *significant-digits*))
3051            (setq average-cats-1-element
3052                  (round-to-n-digits
3053                   (/ (float summed-cats-1-elements) (float in-both-count)) *significant-digits*))
3054            (setq average-cats-2-element
3055                  (round-to-n-digits
3056                   (/ (float summed-cats-2-elements) (float in-both-count)) *significant-digits*))))
3057
3058      ;;PRINTING STARTS
3059      (smart-format output-file "~%;;;~A and ~A ~%;;;compared in terms of ~s~3%"
3060                    catrecs-sorted-file-1 catrecs-sorted-file-2 element)
3061
3062      (if (equal element 'cat-unigram-probability)
3063          (smart-format output-file "~2%(relative-entropy ~s)"
3064                        relative-entropy))
3065      (smart-format output-file "~2%(average-~s-corpus-1 ~s)" element average-cats-1-element)
3066      (smart-format output-file "~2%(average-~s-corpus-2 ~s)" element average-cats-2-element)
3067      (smart-format output-file "~2%(average-~s-difference  ~s)"
3068                    element average-in-both-differences)
3069
3070      (smart-format output-file "~2%(in-both-cats-count ~s)"
3071                    (length in-both-difference-records))
3072      (smart-format output-file "~2%(in-both-difference-records ~s)" in-both-difference-records)
3073
3074      (smart-format output-file "~2%(in-corpus-1-but-not-corpus-2-count ~s)"
3075                    (length in-cats-1-but-not-in-cats-2))
3076      (smart-format output-file "~2%(in-corpus-1-but-not-corpus-2 ~s)" in-cats-1-but-not-in-cats-2)
3077
3078      (smart-format output-file "~2%(in-corpus-2-but-not-corpus-1-count ~s)"
3079                    (length in-cats-2-but-not-in-cats-1))
3080      (smart-format output-file "~2%(in-corpus-2-but-not-corpus-1 ~s)" in-cats-2-but-not-in-cats-1)))
3081
3082    ;;;=============================================================================
3083    ;;;  II.E.6.  MAKING CATRECS.INDEXED
3084    ;;;=============================================================================
3085
3086    (defun make-catrecs-indexed-with-files (directory corpus-prefix)
3087      (let* ((catrecs-sorted-file (format nil "~A~A.catrecs.sorted" directory corpus-prefix))
3088             (catrecs-indexed-file (format nil "~A~A.catrecs.indexed" directory corpus-prefix))
3089             (catrecs-sorted (read-from-file catrecs-sorted-file))
3090             (catrecs-indexed (make-catrecs-indexed catrecs-sorted '(root nil))))
3091        (write-to-file-no-pp catrecs-indexed catrecs-indexed-file)))
3092
3093    ;;;NON-DESTRUCTIVELY alters CATRECS-INDEXED, initially '(root nil).
3094    (defun make-catrecs-indexed (catrecs-sorted catrecs-indexed)
3095      (loop for catrec in catrecs-sorted do
3096            (let ((code (get-catrec-element catrec 'code)))
3097              (setq catrecs-indexed
3098                    (insert-catrec-into-catrecs-indexed catrec code catrecs-indexed))))
3099      ;;Sorting by OBJECT field by default.
3100      (setq catrecs-indexed (resort-catrecs-indexed catrecs-indexed))
3101      catrecs-indexed)
3102
3103    ;;;NON-DESTRUCTIVELY alters CATRECS-INDEXED, initially '(root nil).
3104    (defun insert-catrec-into-catrecs-indexed (catrec code catrecs-indexed)
3105      (let* ((ancestor-codes-top-down (append (reverse (get-code-ancestors code)) (list code)))
3106             (current-node catrecs-indexed))
3107        (loop for object in ancestor-codes-top-down do
3108              (let ((lower-node
3109                     (assoc object (get-node-structure-element current-node 'daughters) :test #'equal)))
3110
3111                (if lower-node
3112                    (setq current-node lower-node)
3113                    (let* ((new-lower-node (make-node-structure object))
3114                           (new-current-node
3115                            ;;NON-DESTRUCTIVELY add new-lower-node to daughters of current-node.
3116                            (put-node-structure-element
3117                             current-node
```

LINE # TEXT

```
3118                          'daughters
3119                          (cons new-lower-node
3120                                (get-node-structure-element current-node 'daughters)))))
3121                    (setq catrecs-indexed
3122                          (subst new-current-node current-node catrecs-indexed))
3123                    (setq current-node new-lower-node)))))
3124
3125      (let ((new-current-node
3126             (put-node-structure-element current-node 'catrec catrec)))
3127        (setq catrecs-indexed
3128              (subst new-current-node current-node catrecs-indexed)))
3129      catrecs-indexed))
3130
3131  ;;;============================================================================
3132  ;;;  II.E.7.  SORTING CATRECS-INDEXED
3133  ;;;============================================================================
3134
3135  ;;;For entire CATRECS-INDEXED, sort each set of daughters recursively.
3136  (defun resort-catrecs-indexed
3137      (catrecs-indexed &key (resort-by 'object))
3138    (let* ((daughters
3139            (get-node-structure-element catrecs-indexed 'daughters)))
3140      (if daughters
3141
3142          ;;There are daughters, so sort them and then recurse on each daughter.
3143          ;;If no daughters, do nothing.
3144          (let ((sorted-daughters
3145                 (sort-node-structures daughters resort-by)))
3146            (setq catrecs-indexed
3147                  (subst sorted-daughters daughters catrecs-indexed))
3148            (loop for daughter in sorted-daughters do
3149                  (let ((new-daughter (resort-catrecs-indexed daughter :resort-by resort-by)))
3150                    (setq catrecs-indexed
3151                          (subst new-daughter daughter catrecs-indexed))))))
3152      catrecs-indexed))
3153
3154  ;;;Puts the node with the highest code first.
3155  (defun sort-node-structures (node-structures element-name)
3156    (let ((position (get-node-structure-element-position element-name)))
3157      ;;LAMBDA means "test whether list-1 is greater than list-2
3158      ;;with respect to the elements at nth position"
3159      (sort node-structures
3160            #'(lambda (list1 list2)
3161                (cond ((numberp (nth position list1))
3162                       (> (nth position list1)(nth position list2)))
3163                      ((stringp (nth position list1))
3164                       (string> (nth position list1)(nth position list2)))
3165                      (t
3166                       (error "Can't sort: pointer to node-structure element which is neither number nor string."))))))))
3167
3168  ;;;============================================================================
3169  ;;;  II.E.8.  QUERYING CATRECS-INDEXED: INCLUDES FINDING WINDOW MATES
3170  ;;;============================================================================
3171
3172  (defvar *CATRECS-INDEXED* nil)
3173
3174  (defun load-catrecs-indexed
3175      (catrecs-indexed-file)
3176    (setq *CATRECS-INDEXED*
3177          (read-from-file catrecs-indexed-file))
3178    (format t "~%LOADING COMPLETED"))
3179
3180  ;;;Gets a catrec from current node in CATRECS-INDEXED.
3181  (defun get-catrec (cat current-node)
3182    (let ((ancestor-codes-top-down (append (reverse (get-code-ancestors cat)) (list cat))))
3183      (get-catrec-using-path ancestor-codes-top-down current-node)))
3184
3185  ;;;Gets a catrec from current node in CATRECS-INDEXED using the full cat ancestor path.
3186  (defun get-catrec-using-path (cat-path current-node)
3187    (let* ((first-ancestor (first cat-path))
3188           (rest-ancestors (rest cat-path))
3189           (daughters (get-node-structure-element current-node 'daughters))
3190           (assoc-result (assoc first-ancestor daughters :test #'equal)))
3191      (if (null rest-ancestors)
3192          (get-node-structure-element assoc-result 'catrec)
3193          (get-catrec-using-path rest-ancestors assoc-result))))
3194
3195  (defun get-cat-hit-count (cat catrecs-indexed)
3196    (let* ((catrec (get-catrec cat catrecs-indexed))
3197           (hit-count (get-catrec-element catrec 'hit-count)))
3198      (if hit-count hit-count 0)))
3199
3200  (defun get-cat-segments (cat catrecs-indexed)
3201    (let* ((catrec (get-catrec cat catrecs-indexed))
3202           (segments (get-catrec-element catrec 'segments)))
3203      (if segments segments nil)))
3204
3205  (defun cat-occurs-in-segment? (cat segment catrecs-indexed)
3206    (member segment (get-cat-segments cat catrecs-indexed)))
3207
3208  ;;;Also used as a predicate to test whether there are ANY such occurrences.
3209  (defun get-cat-occurrences-within-n-segments
3210      (cat reference-segment n catrecs-indexed discourse-final-segments)
3211    (let ((result nil)
3212          (cat-segments
3213           (get-cat-segments cat catrecs-indexed)))
3214      (loop for segment in cat-segments do
3215            (if (and (>= segment reference-segment)
3216                     (<= (- segment reference-segment) n)
3217                     (in-same-discourse? segment reference-segment discourse-final-segments))
3218                (setq result (cons segment result))))
3219      (reverse result)))
3220
3221  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
3222  ;;;  II.F. CAT-COOCRECS
3223  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
3224
3225  ;;;============================================================================
3226  ;;;  II.F.1.  DEFINING CAT-COOCREC DATASTRUCTURE
3227  ;;;============================================================================
3228
3229  (defvar *cat-coocrec-elements*
3230    '((n 0)
3231      (level 1)
3232      (cat-1 2)
3233      (cat-2 3)
3234      (cat-1-segments-with-2-in-range-count 4)
3235      (cat-1-segments-count 5)
3236      (conditional-probability 6)
3237      (mutual-information 7)
```

```
3238      (conditional-probability-smoothed 8)
3239      (mutual-information-smoothed 9)))
3240
3241  ;;;never called
3242  ;(defun make-cat-coocrec (n level cat-1 cat-2)
3243  ;  (let ((cat-coocrec
3244  ;         (loop for element in *cat-coocrec-elements* collect nil)))
3245  ;    ;;<<put-cat-coocrec-element not defined
3246  ;    (setq cat-coocrec (put-cat-coocrec-element cat-coocrec 'n n))
3247  ;    (setq cat-coocrec (put-cat-coocrec-element cat-coocrec 'level level))
3248  ;    (setq cat-coocrec (put-cat-coocrec-element cat-coocrec 'cat-1 cat-1))
3249  ;    (setq cat-coocrec (put-cat-coocrec-element cat-coocrec 'cat-2 cat-2))
3250  ;    cat-coocrec))
3251
3252  (defun get-cat-coocrec-element-position (element-name)
3253    (second
3254      (assoc element-name *cat-coocrec-elements*)))
3255
3256  (defun get-cat-coocrec-element (cat-coocrec element-name)
3257    (let ((position (get-cat-coocrec-element-position element-name)))
3258      (nth position cat-coocrec)))
3259
3260  ;;;============================================================================
3261  ;;;  II.F.2.  MAKING CAT.COOCRECS.INDEXED
3262  ;;;============================================================================
3263
3264  (defun make-cat-coocrecs-indexed-with-files
3265      (n-range pos-1-list pos-2-list directory corpus-prefix morphs-to-cats-prefix
3266               &key
3267               (threshold 0.1)
3268               (threshold-key 'conditional-probability)
3269               (thesaurus 'thesaurus-full))
3270    (let* ((morphrecs-indexed-file (format nil "~A~A.morphrecs.indexed"  directory corpus-prefix))
3271           (catrecs-indexed-file (format nil "~A~A.catrecs.indexed"  directory corpus-prefix))
3272           (morphs-to-cats-file (format nil "~A~A.morphs.to.cats"  directory morphs-to-cats-prefix))
3273           (morphs-global-file (format nil "~A~A.morphs.global"  directory corpus-prefix))
3274           (cat-coocrecs-indexed-file (format nil "~A~A.cat.coocrecs.indexed"  directory corpus-prefix))
3275           (catrecs-indexed (read-from-file catrecs-indexed-file))
3276           (morphrecs-indexed (read-from-file morphrecs-indexed-file))
3277           (morphs-to-cats (read-from-file morphs-to-cats-file))
3278           (cat-coocrecs-indexed
3279            (make-cat-coocrecs-indexed
3280             n-range pos-1-list pos-2-list morphrecs-indexed catrecs-indexed morphs-to-cats morphs-global-file
3281             :threshold threshold :threshold-key threshold-key :thesaurus thesaurus)))
3282      (write-to-file-no-pp cat-coocrecs-indexed cat-coocrecs-indexed-file)))
3283
3284  ;;;Makes CAT-COOCRECS-INDEXED, given CATRECS-INDEXED as input.
3285  ;;;Dynamically filters records below a threshold.
3286  (defun make-cat-coocrecs-indexed
3287      (n-range pos-1-list pos-2-list morphrecs-indexed catrecs-indexed morphs-to-cats morphs-global-file
3288               &key (threshold 0.1)(threshold-key 'conditional-probability)(thesaurus 'thesaurus-full))
3289
3290    (let* ((discourse-final-segments (get-global-value-from-file morphs-global-file 'discourse-final-segments))
3291           (all-segments-count (get-global-value-from-file  morphs-global-file 'all-segments-count))
3292           (cats-for-pos-1-list (get-cats-for-pos-list pos-1-list morphrecs-indexed morphs-to-cats :thesaurus thesaurus))
3293           (cats-for-pos-1-list-by-level (sort-cats-by-level cats-for-pos-1-list))
3294           (cats-for-pos-2-list (get-cats-for-pos-list pos-2-list morphrecs-indexed morphs-to-cats :thesaurus thesaurus))
3295           (cats-for-pos-2-list-by-level (sort-cats-by-level cats-for-pos-2-list))
3296           (levels nil))
3297
3298      ;;Let LEVELS be the union of levels seen for either pos-list.
3299      (loop for level in cats-for-pos-1-list-by-level do
3300            (let ((level-number (first level)))
3301              (setq levels (adjoin level-number levels :test #'equal))))
3302      (loop for level in cats-for-pos-2-list-by-level do
3303            (let ((level-number (first level)))
3304              (setq levels (adjoin level-number levels :test #'equal))))
3305
3306      (loop for n from (first n-range) to (second n-range) collect
3307            (progn (format t "Handling n = ~s~%" n)
3308                   (list n
3309                         (loop for level in levels collect
3310                               (let ((cats-for-pos-1-this-level
3311                                      (second (assoc level cats-for-pos-1-list-by-level))))
3312                                 ;;cat-for-pos-1-list in cats-for-pos-1-list collect
3313                                 (format t "  Handling level = ~s~%" level)
3314                                 (list level
3315                                       (loop for cat-for-pos-1-this-level in cats-for-pos-1-this-level collect
3316                                             (list cat-for-pos-1-this-level
3317                                                   (let ((cats-for-pos-2-this-level
3318                                                          (second (assoc level cats-for-pos-2-list-by-level))))
3319                                                     ;;<<Note use of IT loop construct. See Steele 739.
3320                                                     (loop for cat-for-pos-2-this-level in cats-for-pos-2-this-level
3321                                                           if
3322                                                           (and
3323                                                            ;;ignore co-occurrence of cat with itself
3324                                                            (not (equal cat-for-pos-1-this-level
3325                                                                        cat-for-pos-2-this-level))
3326                                                            (make-cat-coocrec-tail
3327                                                             cat-for-pos-1-this-level
3328                                                             cat-for-pos-2-this-level
3329                                                             n catrecs-indexed
3330                                                             discourse-final-segments
3331                                                             all-segments-count
3332                                                             :threshold threshold
3333                                                             :threshold-key threshold-key))
3334                                                           collect it)))))))))))
3335
3336  ;;;Get cats which have morphs whose pos is in POS-LIST.
3337  (defun get-cats-for-pos-list (pos-list morphrecs-indexed morphs-to-cats &key (thesaurus 'thesaurus-full))
3338    (let ((output nil))
3339      (loop for pos in pos-list do
3340            (let* ((morphrecs-for-pos (get-morphrecs-for-pos pos morphrecs-indexed)))
3341              (loop for morphrec in morphrecs-for-pos do
3342                    (let* ((lex (get-morphrec-element morphrec 'lex))
3343                           (pos (get-morphrec-element morphrec 'pos))
3344                           (cat-codes
3345                            (get-morph-cats lex pos morphs-to-cats :thesaurus thesaurus)))
3346                      (loop for cat-code in cat-codes do
3347                            (setq output
3348                                  (adjoin cat-code output :test #'equal)))))))))
3349
3350      ;;Add ancestors for all the codes -- these are not included in morphs-to-cats.
3351      (loop for cat in output do
3352            (let ((cat-ancestors (get-code-ancestors cat)))
3353              (loop for cat-ancestor in cat-ancestors do
3354                    (setq output (adjoin cat-ancestor output :test #'equal)))))
3355      output))
3356
3357  (defun sort-cats-by-level (thesaurus-codes)
```

```
3358    (let ((output nil))
3359      (loop for thesaurus-code in thesaurus-codes do
3360            (let* ((length (length thesaurus-code))
3361                   (length-group (assoc length output :test #'equal))
3362                   (new-length-group nil)
3363                   (cats-in-length-group (second length-group))
3364                   (new-cats-in-length-group nil))
3365              (if length-group
3366                  (progn
3367                    (setq new-cats-in-length-group (adjoin thesaurus-code cats-in-length-group :test #'equal))
3368                    (setq new-length-group (list length new-cats-in-length-group))
3369                    (setq output
3370                          (subst new-length-group length-group output)))
3371                  (setq output
3372                        (cons (list length (list thesaurus-code))
3373                              output)))))
3374      (setq output (sort output #'< :key #'first))
3375      output))
3376
3377  ;;;============================================================================================
3378  ;;;  II.F.3.  MAKE-CAT-COOCREC-TAIL
3379  ;;;============================================================================================
3380
3381  ;;;Compare MAKE-MORPH-COOCREC-TAIL above.
3382  (defun make-cat-coocrec-tail
3383      (cat-1 cat-2 n catrecs-indexed discourse-final-segments all-segments-count
3384             &key (threshold 0.1)(threshold-key 'conditional-probability))
3385    (let* ((catrec-1 (get-catrec cat-1 catrecs-indexed))
3386           ;;SEGMENTS CONTAINING CAT-1 AT LEAST ONCE.
3387           (cat-1-segments
3388            (get-catrec-element catrec-1 'segments))
3389           ;;NUMBER OF SEGMENTS CONTAINING CAT-1 AT LEAST ONCE.
3390           (cat-1-segments-count (length cat-1-segments))
3391           ;;PROBABILITY THAT CAT-1 OCCURS IN ANY GIVEN SEGMENT.
3392           ;;Pr(a) = CAT-1-SEGMENTS-COUNT/ALL-SEGMENTS-COUNT
3393           (cat-1-occur-in-segment-probability
3394            (get-catrec-element catrec-1 'occur-in-segment-probability))
3395
3396           ;;SMOOTHED PROBABILITY THAT CAT-1 OCCURS IN ANY GIVEN SEGMENT.
3397           ;;Pr(a) smoothed
3398           (cat-1-occur-in-segment-probability-smoothed
3399            (get-catrec-element catrec-1 'occur-in-segment-probability-smoothed))
3400
3401           (cat-1-segments-with-2-in-range-count 0)
3402           ;;Pr(a and b)
3403           (cat-1-occur-in-segment-with-2-in-range-probability 0)
3404           ;;Pr(a and b) smoothed
3405           (cat-1-occur-in-segment-with-2-in-range-probability-smoothed 0)
3406
3407           (catrec-2 (get-catrec cat-2 catrecs-indexed))
3408
3409           ;;PROBABILITY THAT CAT-2 OCCURS IN ANY GIVEN SEGMENT.
3410           ;; = CAT-2-SEGMENTS-COUNT/ALL-SEGMENTS-COUNT
3411           (cat-2-occur-in-segment-probability (get-catrec-element catrec-2 'occur-in-segment-probability))
3412           (cat-2-occur-in-segment-probability-smoothed (get-catrec-element catrec-2 'occur-in-segment-probability-smoothed))
3413
3414           ;;PROBABILITY THAT CAT-2 OCCURS IN ANY GIVEN WINDOW OF WIDTH N PLUS 1.
3415           ;;Pr(b)
3416           (cat-2-occur-in-window-probability (* (float (+ n 1)) cat-2-occur-in-segment-probability))
3417           (cat-2-occur-in-window-probability-smoothed (* (float (+ n 1)) cat-2-occur-in-segment-probability-smoothed)))
3418
3419      ;;Get CAT-1-SEGMENTS-WITH-2-IN-RANGE-COUNT.
3420      (loop for cat-1-segment in cat-1-segments do
3421            (if
3422             (get-cat-occurrences-within-n-segments
3423              cat-2 cat-1-segment n catrecs-indexed discourse-final-segments)
3424             (setq cat-1-segments-with-2-in-range-count (1+ cat-1-segments-with-2-in-range-count))))
3425
3426      ;;Get CAT-1-OCCUR-IN-SEGMENT-WITH-2-IN-RANGE-PROBABILITY.
3427      ;;Pr(a and b)
3428      (setq cat-1-occur-in-segment-with-2-in-range-probability
3429            (round-to-n-digits
3430             (/ (float cat-1-segments-with-2-in-range-count)(float all-segments-count)) *significant-digits*))
3431
3432      ;;Get CAT-1-OCCUR-IN-SEGMENT-WITH-2-IN-RANGE-PROBABILITY-SMOOTHED.
3433      ;;Using 1+ smoothing.
3434      ;;Pr(a and b) smoothed
3435      (setq cat-1-occur-in-segment-with-2-in-range-probability-smoothed
3436            (round-to-n-digits
3437             (/ (+ (float cat-1-segments-with-2-in-range-count)(float 1))
3438                (+ (float 2)(float all-segments-count))) *significant-digits*))
3439
3440      ;;Compute conditional-probability that CAT-2 will occur within n segments of segment S,
3441      ;;given that cat-1 occurs in S.
3442      ;;Return record iff CONDITIONAL-PROBABILITY >= THRESHOLD. Else return nil.
3443      ;;Pr (b given a) = Pr (a and  b)/ Pr (a)
3444      (let* ((conditional-probability
3445              (if (or (zerop cat-1-occur-in-segment-with-2-in-range-probability)
3446                      (zerop cat-1-segments-count))
3447                  ;;<<Giving zero in case denominator is 0.0.
3448                  0.0
3449                  (round-to-n-digits
3450                   (/ (float cat-1-occur-in-segment-with-2-in-range-probability)
3451                      (float cat-1-occur-in-segment-probability)) *significant-digits*)))
3452
3453             ;;log P(b given a) over P(b)
3454             (mutual-information
3455              (if (or (zerop conditional-probability)
3456                      (zerop cat-2-occur-in-window-probability))
3457                  ;;<<Giving zero in case denominator is 0.0.
3458                  0.0
3459                  (round-to-n-digits
3460                   (log (/ (float conditional-probability)
3461                           (float cat-2-occur-in-window-probability)) 2) *significant-digits*)))
3462
3463             ;;As above, but with smoothed nominators and divisors.
3464             (conditional-probability-smoothed
3465              (if (or (zerop cat-1-occur-in-segment-with-2-in-range-probability-smoothed)
3466                      (zerop cat-1-segments-count))
3467                  ;;<<Giving zero in case denominator is 0.0.
3468                  0.0
3469                  (round-to-n-digits
3470                   (/ (float cat-1-occur-in-segment-with-2-in-range-probability-smoothed)
3471                      (float cat-1-occur-in-segment-probability-smoothed)) *significant-digits*)))
3472
3473             (mutual-information-smoothed
3474              (if (or (zerop conditional-probability-smoothed)
3475                      (zerop cat-2-occur-in-window-probability))
3476                  ;;<<Giving zero in case denominator is 0.0.
3477                  0.0
```

LINE # TEXT

```
3478                  (round-to-n-digits
3479                   (log (/ (float conditional-probability-smoothed)
3480                           (float cat-2-occur-in-window-probability-smoothed)) 2) *significant-digits*)))
3481             (threshold-key-value
3482              (case threshold-key
3483                ((conditional-probability)
3484                 conditional-probability)
3485                ((conditional-probability-smoothed)
3486                 conditional-probability-smoothed)
3487                ((mutual-information)
3488                 mutual-information)
3489                ((mutual-information-smoothed)
3490                 mutual-information-smoothed))))
3491
3492        (if (>= threshold-key-value threshold)
3493            (list cat-2
3494                  cat-1-segments-with-2-in-range-count
3495                  cat-1-segments-count
3496                  conditional-probability
3497                  mutual-information
3498                  conditional-probability-smoothed
3499                  mutual-information-smoothed)))))
3500
3501  ;;;============================================================================
3502  ;;;  II.F.4.  QUERYING CAT-COOCRECS-INDEXED
3503  ;;;============================================================================
3504
3505  (defvar *CAT-COOCRECS-INDEXED* nil)
3506
3507  (defun load-cat-coocrecs-indexed
3508      (cat-coocrecs-indexed-file)
3509    (setq *CAT-COOCRECS-INDEXED*
3510          (read-from-file cat-coocrecs-indexed-file))
3511    (format t "~%LOADING COMPLETED"))
3512
3513  ;;;Gets from CAT-COOCRECS-INDEXED the conditional probability of occurrence within N segments
3514  ;;;of cat-2 given cat-1.
3515  (defun get-cat-conditional-probability
3516      (n level cat-1 cat-2 cat-coocrecs-indexed)
3517    (get-cat-coocrec-element
3518     (make-and-get-cat-coocrec
3519      n level cat-1 cat-2 cat-coocrecs-indexed)
3520     'conditional-probability))
3521
3522  ;;;Makes and gets (fetches) CAT-COOCREC.
3523  ;;;Fetches partial cat-coocrec, which is terminal of CAT-COOCRECS-INDEXED.
3524  ;;;Completes it, creating and returning full CAT-COOCREC.
3525  (defun make-and-get-cat-coocrec
3526      (n level cat-1 cat-2 cat-coocrecs-indexed)
3527    (let* ((n-group (second (assoc n cat-coocrecs-indexed)))
3528           (level-group (second (assoc level n-group)))
3529           (cat-1-group (second (assoc cat-1 level-group :test #'equal)))
3530           (cat-2-record (assoc cat-2 cat-1-group :test #'equal)))
3531      (if cat-2-record
3532          ;;<<CREATION OF CAT-COOCREC
3533          ;;<<avoiding MAKE-CAT-COOCREC to avoid clumsy copying
3534          (append (list n level cat-1) cat-2-record)
3535          (append (list n level cat-1 cat-2) '(0.0 0.0 0.0 0.0 0.0 0.0 0.0)))))
3536
3537  ;;;============================================================================
3538  ;;;  II.F.5.  CAT-WINDOW-MATE QUERY FUNCTIONS
3539  ;;;============================================================================
3540
3541  ;;;Compare GET-MORPH-WINDOW-MATES-WITH-FILES, above.
3542  (defun get-cat-window-mates-with-files
3543      (n cat-1 directory corpus-prefix &key
3544         (threshold nil)(threshold-key 'conditional-probability)
3545         (filter-self t)(db-pos-display 'english)(return-cats-only nil))
3546    (ignore db-pos-display)
3547    (let* ((cat-coocrecs-indexed-file (format nil "~A~A.cat.coocrecs.indexed" directory corpus-prefix)))
3548      (if (null *cat-coocrecs-indexed*)
3549          (progn
3550            (format t "~%Loading cat-coocrecs-indexed from file ~a." cat-coocrecs-indexed-file)
3551            (load-cat-coocrecs-indexed cat-coocrecs-indexed-file)))
3552      (let* ((output nil)
3553             (cat-1-level (length cat-1))
3554             (n-group (second (assoc n *cat-coocrecs-indexed*)))
3555             (level-group (second (assoc cat-1-level n-group)))
3556             (cat-1-group (second (assoc cat-1 level-group :test #'equal)))
3557             (cat-1-group-cat-coocrecs
3558              (loop for partial-cat-coocrec in cat-1-group collect
3559                    (append (list n cat-1-level cat-1) partial-cat-coocrec)))
3560             (window-mates-including-self
3561              (if threshold
3562                  (loop for cat-coocrec in cat-1-group-cat-coocrecs when
3563                        (>= (get-cat-coocrec-element cat-coocrec threshold-key) threshold)
3564                        collect cat-coocrec)
3565                  cat-1-group-cat-coocrecs)))
3566        (if filter-self
3567            (let ((cat-coocrec-self
3568                   (make-and-get-cat-coocrec
3569                    n cat-1-level cat-1 cat-1 'cat-coocrecs-indexed*)))
3570
3571              (setq output (remove cat-coocrec-self window-mates-including-self)))
3572            (setq output window-mates-including-self))
3573        (if return-cats-only
3574            (let ((temp-output nil))
3575              (loop for cat-coocrec in output do
3576                    (let ((cat-2 (get-cat-coocrec-element cat-coocrec 'cat-2)))
3577                      (setq temp-output
3578                            (adjoin cat-2 temp-output :test #'equal))))
3579              (setq output temp-output)))
3580        output)))
3581
3582  ;;;============================================================================
3583  ;;;  II.F.6.  MAKING CAT.COOCRECS.SORTED
3584  ;;;============================================================================
3585
3586  (defun make-cat-coocrecs-sorted-with-files
3587      (directory corpus-prefix &key (threshold 0.1)(threshold-key 'conditional-probability))
3588    (let* ((cat-coocrecs-indexed-file (format nil "~A~A.cat.coocrecs.indexed" directory corpus-prefix))
3589           (cat-coocrecs-sorted-file (format nil "~A~A.cat.coocrecs.sorted" directory corpus-prefix))
3590           (cat-coocrecs-indexed (read-from-file cat-coocrecs-indexed-file))
3591           (cat-coocrecs-sorted
3592            (make-cat-coocrecs-sorted
3593             cat-coocrecs-indexed
3594             :threshold threshold
3595             :threshold-key threshold-key)))
3596      (write-to-file-no-pp cat-coocrecs-sorted cat-coocrecs-sorted-file)))
3597
```

LINE # TEXT

```
(defun make-cat-coocrecs-sorted
    (cat-coocrecs-indexed &key (threshold 0.1)(threshold-key 'conditional-probability))
  (let (output)
    (loop for n-group in cat-coocrecs-indexed do
          (loop for level-group in (second n-group) do
                (loop for cat-1-group in (second level-group) do
                      (loop for partial-cat-coocrec in (second cat-1-group) do
                            ;;<<convert format later
                            (let* ((n (first n-group))
                                   (level (first level-group))
                                   (cat-1 (first cat-1-group))
                                   (cat-2 (first partial-cat-coocrec))
                                   (cat-coocrec
                                    ;;<<CREATION OF CAT-COOCREC.
                                    ;;<<CONVERTS PARTIAL CAT-COOCREC TO COMPLETE ONE AT FETCH TIME.
                                    (make-and-get-cat-coocrec n level cat-1 cat-2 cat-coocrecs-indexed))
                                   (threshold-key-value
                                    (get-cat-coocrec-element
                                     cat-coocrec 'conditional-probability)))
                              (if (>= threshold-key-value threshold)
                                  (setq output (cons cat-coocrec output))))))))
    ;;<<Sorting by mutual-information-smoothed -- fixed key.
    (sort-cat-coocrecs-sorted output threshold-key)))

;;;============================================================================
;;;  II.F.7.   SORTING CAT-COOCRECS-SORTED
;;;============================================================================

(defun sort-cat-coocrecs-sorted (cat-coocrecs-sorted element-name)
  (let ((position (get-cat-coocrec-element-position element-name)))
    ;;lambda means "test whether list-1 is greater than list-2
    ;;with respect to the elements at nth position"
    (sort cat-coocrecs-sorted
          #'(lambda (list1 list2)
              (if (not (numberp (nth position list1)))
                  (error "Can't sort: pointer to non-numerical element in cat-coocrec.")
                  (> (nth position list1)(nth position list2)))))))

;;;============================================================================
;;;  II.F.8.  QUERYING CAT-COOCRECS-SORTED
;;;============================================================================

(defvar *CAT-COOCRECS-SORTED* nil)

(defun load-cat-coocrecs-sorted
    (cat-coocrecs-sorted-file)
  (setq *CAT-COOCRECS-SORTED*
        (read-from-file cat-coocrecs-sorted-file))
  (format t "~%LOADING COMPLETED"))

;;;Collects list of e.g. CONDITIONAL-PROBABILITY values over all CAT-COOCRECS-SORTED.
(defun get-all-cat-coocrec-elements
    (element-name cat-coocrecs-sorted)
  (loop for cat-coocrec in cat-coocrecs-sorted collect
        (get-cat-coocrec-element cat-coocrec element-name)))

;;;============================================================================
;;;  II.F.9.  DEFINING CAT-COOCREC-DIFFREC
;;;============================================================================

;;;A CAT-COOCREC-DIFFREC is a record of the difference between corresponding cat-coocrecs
;;;in two corpora.
(defvar *cat-coocrec-diffrec-elements*
  '((n 0)
    (level 1)
    (cat-1 2)
    (cat-2 3)
    (element-1 4)
    (element-2 5)
    (difference 6)
    (count-1 7)                         ;cat-1 segment count in corpus-1
    (count-2 8)                         ;cat-1 segment count in corpus-2
    (count-sum 9)))

(defun make-cat-coocrec-diffrec (n level cat-1 cat-2 element-1 element-2 difference count-1 count-2 count-sum)
  (let ((cat-coocrec-diffrec
         (loop for element in *cat-coocrec-diffrec-elements* collect nil)))
    (setq cat-coocrec-diffrec (put-cat-coocrec-diffrec-element cat-coocrec-diffrec 'n n))
    (setq cat-coocrec-diffrec (put-cat-coocrec-diffrec-element cat-coocrec-diffrec 'level level))
    (setq cat-coocrec-diffrec (put-cat-coocrec-diffrec-element cat-coocrec-diffrec 'cat-1 cat-1))
    (setq cat-coocrec-diffrec (put-cat-coocrec-diffrec-element cat-coocrec-diffrec 'cat-2 cat-2))
    (setq cat-coocrec-diffrec (put-cat-coocrec-diffrec-element cat-coocrec-diffrec 'element-1 element-1))
    (setq cat-coocrec-diffrec (put-cat-coocrec-diffrec-element cat-coocrec-diffrec 'element-2 element-2))
    (setq cat-coocrec-diffrec (put-cat-coocrec-diffrec-element cat-coocrec-diffrec 'difference difference))
    (setq cat-coocrec-diffrec (put-cat-coocrec-diffrec-element cat-coocrec-diffrec 'count-1 count-1))
    (setq cat-coocrec-diffrec (put-cat-coocrec-diffrec-element cat-coocrec-diffrec 'count-2 count-2))
    (setq cat-coocrec-diffrec (put-cat-coocrec-diffrec-element cat-coocrec-diffrec 'count-sum count-sum))
    cat-coocrec-diffrec))

(defun get-cat-coocrec-diffrec-element-position (element-name)
  (second
   (assoc element-name *cat-coocrec-diffrec-elements*)))

(defun get-cat-coocrec-diffrec-element (cat-coocrec-diffrec element-name)
  (let ((position (get-cat-coocrec-diffrec-element-position element-name)))
    (nth position cat-coocrec-diffrec)))

(defun put-cat-coocrec-diffrec-element (cat-coocrec-diffrec element-name value)
  (let ((cat-coocrec-diffrec-copy (copy-list cat-coocrec-diffrec))
        (position (get-cat-coocrec-diffrec-element-position element-name)))
    (setf (nth position cat-coocrec-diffrec-copy) value)
    cat-coocrec-diffrec-copy))

(defun add-cat-coocrec-diffrec-element (cat-coocrec-diffrec element-name value)
  (let ((cat-coocrec-diffrec-copy (copy-list cat-coocrec-diffrec))
        (old-value (get-cat-coocrec-diffrec-element cat-coocrec-diffrec element-name))
        (new-value nil)
        (position (get-cat-coocrec-diffrec-element-position element-name)))
    (if (not (listp old-value))
        (error "old value not a list")
        (progn (setq new-value (adjoin value old-value :test #'equal))
               (setf (nth position cat-coocrec-diffrec-copy) new-value)))
    cat-coocrec-diffrec-copy))

(defun sort-cat-coocrec-diffrecs (cat-coocrec-diffrecs element-name)
  (let ((position (get-cat-coocrec-diffrec-element-position element-name)))
    ;;LAMBDA means "test whether list-1 is greater than list-2
    ;;with respect to the elements at nth position"
    (sort cat-coocrec-diffrecs
          #'(lambda (list1 list2)
```

```
LINE #  TEXT
3718                (if (not (numberp (nth position list1)))
3719                    (error "can't sort: pointer to non-numerical element in rec")
3720                    (> (nth position list1)(nth position list2)))))))
3721
3722  ;;;============================================================================
3723  ;;;  II.F.10.  COMPARING CAT-COOCRECS-SORTED
3724  ;;;============================================================================
3725
3726  ;;;Compare CAT-COOCRECS-SORTED in two corpora.
3727  ;;;Assume CAT-COOCRECS-SORTED-FILES have been presorted as desired.
3728  (defun compare-cat-coocrecs-sorted
3729      (directory corpus-prefix-1 corpus-prefix-2
3730                 &key (element 'conditional-probability))
3731
3732    (let* ((cat-coocrecs-sorted-file-1 (format nil "~A~A.cat.coocrecs.sorted"  directory corpus-prefix-1))
3733           (cat-coocrecs-indexed-file-2 (format nil "~A~A.cat.coocrecs.indexed"  directory corpus-prefix-2))
3734           (cat-coocrecs-sorted-file-2 (format nil "~A~A.cat.coocrecs.sorted"  directory corpus-prefix-2))
3735           (output-file (format nil "~A~A.vs.~A.cat.coocrecs.comparison"  directory corpus-prefix-1 corpus-prefix-2))
3736           (cat-coocrecs-1 (read-from-file cat-coocrecs-sorted-file-1))
3737           (pairs-1
3738            (loop for record in cat-coocrecs-1 collect
3739                  (list
3740                   (get-cat-coocrec-element record 'n)
3741                   (get-cat-coocrec-element record 'level)
3742                   (get-cat-coocrec-element record 'cat-1)
3743                   (get-cat-coocrec-element record 'cat-2))))
3744           (cat-coocrecs-indexed-2 (read-from-file cat-coocrecs-indexed-file-2))
3745           (cat-coocrecs-2 (read-from-file cat-coocrecs-sorted-file-2))
3746           (pairs-2
3747            (loop for record in cat-coocrecs-2 collect
3748                  (list
3749                   (get-cat-coocrec-element record 'n)
3750                   (get-cat-coocrec-element record 'level)
3751                   (get-cat-coocrec-element record 'cat-1)
3752                   (get-cat-coocrec-element record 'cat-2))))
3753
3754           (in-both (intersection pairs-1 pairs-2 :test #'equal))
3755           (in-both-count (length in-both))
3756           (in-pairs-1-but-not-in-pairs-2 (set-difference pairs-1 pairs-2 :test  #'equal))
3757           (in-pairs-2-but-not-in-pairs-1 (set-difference pairs-2 pairs-1 :test  #'equal))
3758
3759           (in-both-difference-records-corpus-1-order nil)
3760           (in-both-difference-records-least-difference-order nil)
3761           (in-both-difference-records-greatest-count-sum-order nil)
3762
3763           (summed-in-both-differences 0.0)
3764           (average-in-both-differences nil)
3765           (average-pairs-1-element nil)
3766           (average-pairs-2-element nil)
3767           (summed-pairs-1-elements 0.0)
3768           (summed-pairs-2-elements 0.0)
3769
3770           (catrecs-indexed-file-1 (format nil "~A~A.catrecs.indexed"  directory corpus-prefix-1))
3771           (catrecs-indexed-1 (read-from-file catrecs-indexed-file-1))
3772           (catrecs-indexed-file-2 (format nil "~A~A.catrecs.indexed"  directory corpus-prefix-2))
3773           (catrecs-indexed-2 (read-from-file catrecs-indexed-file-2)))
3774
3775      (if output-file
3776          (if (probe-file output-file)
3777              (delete-file output-file)))
3778
3779      ;;Loop through CAT-COOCRECS-SORTED-FILE-1, the training corpus,
3780      ;;so that this ordering will be preserved.
3781      ;;If retrieved info shows a pair which is in both corpora, go ahead.
3782      (loop for record-1 in cat-coocrecs-1 do
3783            (let ((n (get-cat-coocrec-element record-1 'n))
3784                  (level (get-cat-coocrec-element record-1 'level))
3785                  (cat-1 (get-cat-coocrec-element record-1 'cat-1))
3786                  (cat-2 (get-cat-coocrec-element record-1 'cat-2)))
3787              (if (member (list n level cat-1 cat-2) in-both :test #'equal)
3788                  (let* ((record-2
3789                          ;;<<RE-CREATION OF CAT-COOCREC (at fetch time)
3790                          (make-and-get-cat-coocrec
3791                           n level cat-1 cat-2 cat-coocrecs-indexed-2))
3792
3793                         ;;ELEMENT is e.g. CONDITIONAL-PROBABILITY
3794                         (element-1 (get-cat-coocrec-element record-1 element))
3795                         (element-2 (get-cat-coocrec-element record-2 element))
3796                         (difference
3797                          (round-to-n-digits
3798                           (abs (- element-1 element-2)) *significant-digits*)))
3799
3800                    ;;MAKE A NEW DIFFERENCE RECORD.
3801                    (let* ((catrec-1 (get-catrec cat-1 catrecs-indexed-1))
3802                           ;;<<get same info -- for cat-1 -- from catrec-2
3803                           (catrec-2 (get-catrec cat-1 catrecs-indexed-2))
3804                           (count-1 (get-catrec-element catrec-1 'cat-segments-count))
3805                           (count-2 (get-catrec-element catrec-2 'cat-segments-count))
3806                           (count-sum (+ count-1 count-2))
3807                           (new-cat-coocrec-diffrec
3808                            (make-cat-coocrec-diffrec
3809                             n level cat-1 cat-2 element-1 element-2 difference count-1 count-2 count-sum)))
3810
3811                      (setq in-both-difference-records-corpus-1-order
3812                            (cons
3813                             new-cat-coocrec-diffrec
3814                             in-both-difference-records-corpus-1-order)))
3815
3816                    ;;Maintain running sums.
3817                    (setq summed-in-both-differences (+ summed-in-both-differences difference))
3818                    (setq summed-pairs-1-elements (+ summed-pairs-1-elements element-1))
3819                    (setq summed-pairs-2-elements (+ summed-pairs-2-elements element-2))))))
3820
3821      (setq in-both-difference-records-corpus-1-order (reverse in-both-difference-records-corpus-1-order))
3822      (setq in-both-difference-records-least-difference-order
3823            (reverse (sort-cat-coocrec-diffrecs in-both-difference-records-corpus-1-order 'difference)))
3824      (setq in-both-difference-records-greatest-count-sum-order
3825            (sort-cat-coocrec-diffrecs in-both-difference-records-corpus-1-order 'count-sum))
3826
3827      ;;Get averages if any records are shared.
3828      ;;If not, averages remain nil.
3829      (if in-both-count
3830          (progn
3831            (setq average-in-both-differences
3832                  (round-to-n-digits
3833                   (/ (float summed-in-both-differences) (float in-both-count)) *significant-digits*))
3834            (setq average-pairs-1-element
3835                  (round-to-n-digits
3836                   (/ (float summed-pairs-1-elements) (float in-both-count)) *significant-digits*))
3837            (setq average-pairs-2-element
```

```
               (round-to-n-digits
                 (/ (float summed-pairs-2-elements) (float in-both-count)) *significant-digits*))))

     ;;PRINTING STARTS
     (smart-format output-file "~%;;;~A and ~A ~%;;;compared in terms of ~s"
                   cat-coocrecs-sorted-file-1 cat-coocrecs-sorted-file-2 element)
     (smart-format output-file "~2%(average-~s-corpus-1 ~s)" element average-pairs-1-element)
     (smart-format output-file "~2%(average-~s-corpus-2 ~s)" element average-pairs-2-element)
     (smart-format output-file "~2%(average-~s-difference ~s)"
                   element average-in-both-differences)

     (smart-format output-file "~2%(in-both-pairs-count ~s)"
                   (length in-both-difference-records-corpus-1-order))
     (smart-format output-file "~2%(difference-records-corpus-1-order ~s)" in-both-difference-records-corpus-1-order)

     (smart-format output-file "~2%(difference-records-least-difference-order ~s)"
                   in-both-difference-records-least-difference-order)

     (smart-format output-file "~2%(difference-records-greatest-count-sum-order ~s)"
                   in-both-difference-records-greatest-count-sum-order)

     (smart-format output-file "~2%(in-corpus-1-but-not-corpus-2-count ~s)"
                   (length in-pairs-1-but-not-in-pairs-2))
     (smart-format output-file "~2%(in-corpus-1-but-not-corpus-2 ~s)" in-pairs-1-but-not-in-pairs-2)

     (smart-format output-file "~2%(in-corpus-2-but-not-corpus-1-count ~s)"
                   (length in-pairs-2-but-not-in-pairs-1))
     (smart-format output-file "~2%(in-corpus-2-but-not-corpus-1 ~s)" in-pairs-2-but-not-in-pairs-1)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  II.G. THESAURUS, MORPHS-TO-CATS, CATS-TO-MORPHS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;============================================================================
;;;  II.G.1.  DEFINING THESAURUS-ENTRY
;;;============================================================================

;;;<<Unusual format for THESAURUS-FULL requires exceptional treatment.
(defvar *thesaurus-entry-elements-atr*
  '((key 0)
    (kana 1)
    (romaji 2)
    (pos-code 3)
    (cat-codes 4)
    (kanji 5)))

;;;no make- fn needed: read-only

(defun get-thesaurus-entry-element-position-atr (element-name)
  (second
   (assoc element-name *thesaurus-entry-elements-atr*)))

(defun get-thesaurus-entry-element (thesaurus-entry element-name &key (thesaurus 'thesaurus-full))
  (cond ((equal thesaurus 'thesaurus-full)
         (cond ((equal element-name 'key)
                (first thesaurus-entry))
               ((equal element-name 'kana)
                (second thesaurus-entry))
               ((equal element-name 'cat-codes)
                (rest (rest thesaurus-entry)))
               (t
                (error "Bad index into entry in FULL thesaurus."))))
        ((equal thesaurus 'thesaurus-atr)
         (let ((position (get-thesaurus-entry-element-position-atr element-name)))
           (nth position thesaurus-entry)))
        (t
         (error "Bad thesaurus specification to get-thesaurus-entry-element."))))

;;;============================================================================
;;;  II.G.2.  THESAURUS LOAD AND ACCESS
;;;============================================================================

(defvar *thesaurus-atr* nil)

(defvar *thesaurus-full* nil)

(defvar *thesaurus-hierarchy* nil)

(defvar *hierarchy-romaji* nil)

;;;THESAURUS-FILE stores expressions as stream, not list.
(defun load-thesaurus-atr (&key (thesaurus-file "~/survey/jap/topic/ATR"))
  (format t "~%Loading thesaurus from file ~a" thesaurus-file)
  (setq *thesaurus-atr* (read-loop thesaurus-file))
  (format t "~%Finished loading thesaurus from file ~a" thesaurus-file))

(defun load-thesaurus-full (&key (thesaurus-file "~/survey/jap/topic/FULL"))
  (format t "~%Loading thesaurus from file ~a" thesaurus-file)
  (setq *thesaurus-full* (read-loop thesaurus-file))
  (format t "~%Finished loading thesaurus from file ~a" thesaurus-file))

(defun get-thesaurus-entries-with-files
    (morphrec directory &key (thesaurus 'thesaurus-full))
  (cond ((equal thesaurus 'thesaurus-full)
         (if (null *thesaurus-full*)
             (progn
               (format t "~%Loading thesaurus-full.")
               (load-thesaurus-full))))
        ((equal thesaurus 'thesaurus-atr)
         (if (null *thesaurus-atr*)
             (progn
               (format t "~%Loading thesaurus-atr.")
               (load-thesaurus-atr))))
        (t (error "In get-thesaurus-entries-with-files, bad thesaurus specified: ~s." thesaurus)))
  (cond ((equal thesaurus 'thesaurus-full)
         (get-thesaurus-entries-full morphrec directory))
        ((equal thesaurus 'thesaurus-atr)
         (get-thesaurus-entries-atr morphrec directory))
        (t (error "In get-thesaurus-entries-with-files, bad thesaurus specified: ~s." thesaurus))))


;;;<<examples of ATR format.
;;;ENTRIES ARE STRINGS, NOT SYMBOLS
;;;("側" "がわ" "gawa" "04" ("101d" "104f") "側")
;;;("バス" "バス" "basu" "04" "997a" "バス")
;;;("なまえ" "なまえ" "namae" "04" ("822" "822b") "名前")

;;;If morph is listed in thesaurus by kanji, assoc will find it.
;;;Else search through all entries by kana.
;;;There may be several entries with same kana.
```

LINE # TEXT

```
3958 ;;;They should be grouped together, but one error was found.
3959 ;;;Abort kana searches at first non-match after morph has been found.
3960 (defun get-thesaurus-entries-atr (morphrec directory)
3961   (let* ((thesaurus-prefix "atr")
3962          (lex (get-morphrec-element morphrec 'lex))
3963          (lex-string (format nil "~s" lex))
3964          (pos (get-morphrec-element morphrec 'pos))
3965          (atr-pos (map-pos-romaji-to-atr pos)))
3966
3967     ;;If atr-pos is verb, convert to dictionary form.
3968     (if (or (equal pos 'verb)
3969             (equal pos 21)
3970             (equal pos '本動詞))
3971         (progn
3972           (setq lex
3973                 (get-dictionary-form-for-verb-stem
3974                  lex directory thesaurus-prefix))
3975           (setq lex-string (format nil "~s" lex))))
3976
3977     ;;If atr-pos is common-noun, convert to dictionary form.
3978     (if (or (equal pos 'common-noun)
3979             (equal pos 14)
3980             (equal pos '普通名詞))
3981         (setq lex
3982               (get-dictionary-form-for-common-noun
3983                lex directory thesaurus-prefix)))
3984
3985     ;;If morph is kanji, use ASSOC.
3986     (let ((morph-entry (assoc lex-string *thesaurus-atr* :test #'equal))
3987           (candidates nil))
3988
3989       ;;But if not found according to kanji, have to search by kana.
3990       (if (null morph-entry)
3991           (let ((found? nil)
3992                 (passed? nil))
3993
3994             ;;Set found? to t when first match is found.
3995             ;;Set passed? to t at first non-match after that.
3996             (loop for entry in *thesaurus-atr* until passed? do
3997                   (if (and (equal lex-string (second entry))
3998                            (equal atr-pos (fourth entry)))
3999                       (progn
4000                         (setq found? t)
4001                         (setq candidates (cons entry candidates)))
4002                       (if found?
4003                           (setq passed? t))))))
4004       (if morph-entry
4005           (list morph-entry)
4006           candidates))))
4007
4008 (defun get-thesaurus-entries-full (morphrec directory)
4009   (let* ((thesaurus-prefix "full")
4010          (lex (get-morphrec-element morphrec 'lex))
4011          (pos (get-morphrec-element morphrec 'pos)))
4012
4013     ;;If atr-pos is verb, convert to dictionary form.
4014     (if (or (equal pos 'verb)
4015             (equal pos 21)
4016             (equal pos '本動詞))
4017         (setq lex
4018               (get-dictionary-form-for-verb-stem
4019                lex directory thesaurus-prefix)))
4020
4021     ;;If atr-pos is common-noun, convert to dictionary form.
4022     (if (or (equal pos 'common-noun)
4023             (equal pos 14)
4024             (equal pos '普通名詞))
4025         (setq lex
4026               (get-dictionary-form-for-common-noun
4027                lex directory thesaurus-prefix)))
4028
4029     ;;If morph is kanji, use ASSOC.
4030     (let ((morph-entry (assoc lex *thesaurus-full* :test #'equal))
4031           (candidates nil))
4032
4033       ;;But if not found according to kanji, have to search by kana.
4034       (if (null morph-entry)
4035           (let ((found? nil)
4036                 (passed? nil))
4037
4038             ;;Set found? to t when first match is found.
4039             ;;Set passed? to t at first non-match after that.
4040             (loop for entry in *thesaurus-full* until passed? do
4041                   ;;This test neglects pos.
4042                   (if (equal lex (second entry))
4043                       (progn
4044                         (setq found? t)
4045                         (setq candidates (cons entry candidates)))
4046                       (if found?
4047                           (setq passed? t))))))
4048       (if morph-entry
4049          (list morph-entry)
4050          candidates))))
4051
4052 ;;;<<Beware fixed keys: FOURTH, FIFTH.
4053 (defun map-pos-romaji-to-atr (pos-romaji)
4054   (let ((entry nil)
4055         (found? nil))
4056     (loop for x in *pos-table* until found? do
4057           (if (equal (fourth x) pos-romaji)
4058               (progn
4059                 (setq found? t)
4060                 (setq entry x))))
4061     (fifth entry)))
4062
4063 ;;;============================================================================
4064 ;;;  II.G.3.  THESAURUS HIERARCHY LOAD AND ACCESS
4065 ;;;============================================================================
4066
4067 (defun load-hierarchy (&key (hierarchy-file "~/survey/jap/topic/HIERARCHY")
4068                             (hierarchy-romaji-file "~/survey/jap/topic/HIERARCHY.romaji"))
4069   (setq *thesaurus-hierarchy* (read-loop hierarchy-file))
4070   (format t "~%Finished loading hierarchy from file ~a" hierarchy-file)
4071
4072   (if hierarchy-romaji-file
4073       (progn (setq *hierarchy-romaji* (read-loop hierarchy-romaji-file))
4074              (format t "~%Finished loading hierarchy-romaji from file ~a" hierarchy-romaji-file))))
4075
4076 (defun get-hierarchy-entry (code)
4077   (assoc code *thesaurus-hierarchy* :test #'equal))
```

LINE # TEXT

```
4078
4079  (defun get-hierarchy-romaji-entry (code)
4080    (assoc code *hierarchy-romaji* :test #'equal))
4081
4082  (defun get-hierarchy-both-entry (code)
4083    (append (get-hierarchy-entry code)
4084            (rest (get-hierarchy-romaji-entry code))))
4085
4086  ;;;Returns list of ancestor code strings, ordered with most specific first.
4087  (defun get-code-ancestors (code)
4088    (let* ((char-list (coerce code 'list))
4089           (reverse-char-list (reverse char-list))
4090           (result nil))
4091      (loop while reverse-char-list do
4092            (let ((remainder (rest reverse-char-list)))
4093              (if remainder
4094                  (setq result
4095                    (cons (reverse remainder) result)))
4096              (setq reverse-char-list remainder)))
4097      (loop for ancestor in (reverse result) collect
4098            (coerce ancestor 'string))))
4099
4100  ;;;============================================================================
4101  ;;;  II.G.4.  MAKING MORPHS-TO-CATS
4102  ;;;============================================================================
4103
4104  (defun make-morphs-to-cats-with-files
4105      (pos-list directory corpus-prefix &key (thesaurus 'thesaurus-full))
4106    (let ((morphrecs-indexed-file (format nil "~A~A.morphrecs.indexed" directory corpus-prefix))
4107          (morphs-to-cats-file (format nil "~A~A.morphs.to.cats" directory corpus-prefix))
4108          (morphs-to-cats-global-file (format nil "~A~A.morphs.to.cats.global" directory corpus-prefix))
4109          (output nil)
4110          (global-output nil))
4111
4112      ;;Make sure required thesaurus is loaded.
4113      (cond ((and (equal thesaurus 'thesaurus-atr)
4114                  (null *thesaurus-atr*))
4115             (load-thesaurus-atr))
4116            ((and (equal thesaurus 'thesaurus-full)
4117                  (null *thesaurus-full*))
4118             (load-thesaurus-full)))
4119
4120      (loop for pos in pos-list do
4121            (let* ((morphrecs (third (assoc pos (read-from-file morphrecs-indexed-file))))
4122                   (morphrecs-length (length morphrecs)))
4123              (let ((output-this-pos nil)
4124                    (morphrecs-with-multiple-entries-count 0)
4125                    (morphrecs-with-multiple-entries  nil)
4126                    (failures 0)
4127                    (morphrecs-without-entries nil))
4128                (format t "~%Getting thesaurus entries from ~s for all morphrecs with pos ~s.~%" thesaurus pos)
4129                (loop for morphrec in morphrecs do
4130                      (format t ".")
4131                      (let ((entries
4132                             (cond ((equal thesaurus 'thesaurus-atr)
4133                                    (get-thesaurus-entries-atr morphrec directory))
4134                                   ((equal thesaurus 'thesaurus-full)
4135                                    (get-thesaurus-entries-full morphrec directory))
4136                                   (t (error "Bad thesaurus specification in make-morphs-to-cats-with-files.")))))
4137                        (if (null entries)
4138                            ;;No entries were found.
4139                            (progn
4140                              (setq failures (1+ failures))
4141                              ;;Morphrec will appear in output with null thesaurus-entries field.
4142                              (setq output-this-pos (cons morphrec output-this-pos))
4143                              (setq morphrecs-without-entries
4144                                    (cons morphrec morphrecs-without-entries)))
4145
4146                            ;;Entries were found, so update morphrec with those entries.
4147                            (let ((new-morphrec (put-morphrec-element morphrec 'thesaurus-entries entries)))
4148                              (if (second entries) ;more than one entry was found
4149                                  (progn (setq morphrecs-with-multiple-entries
4150                                               (cons new-morphrec morphrecs-with-multiple-entries))
4151                                         (setq morphrecs-with-multiple-entries-count
4152                                               (1+ morphrecs-with-multiple-entries-count))))
4153                              (setq output-this-pos (cons new-morphrec output-this-pos))))))
4154
4155                (let ((failure-ratio (/ (float failures)(float morphrecs-length))))
4156                  (setq output (cons (list pos nil (reverse output-this-pos)) output))
4157
4158                  (setq global-output
4159                        (cons (list pos nil
4160                                    (list `(morphrecs-length ,morphrecs-length)
4161                                          `(failures ,failures)
4162                                          `(failure-ratio ,failure-ratio)
4163                                          `(morphrecs-without-entries ,(reverse morphrecs-without-entries))
4164                                          `(morphrecs-with-multiple-entries-count ,morphrecs-with-multiple-entries-count)
4165                                          `(morphrecs-with-multiple-entries ,(reverse morphrecs-with-multiple-entries))))
4166                              global-output))))))
4167      (write-to-file-no-pp output morphs-to-cats-file)
4168      (write-to-file-no-pp global-output morphs-to-cats-global-file)))
4169
4170  ;;;============================================================================
4171  ;;;  II.G.5.  QUERYING MORPHS-TO-CATS
4172  ;;;============================================================================
4173
4174  (defvar *MORPHS-TO-CATS* nil)
4175
4176  (defun load-morphs-to-cats
4177      (morphs-to-cats-file)
4178    (setq *MORPHS-TO-CATS*
4179          (read-from-file morphs-to-cats-file)))
4180
4181  (defun get-morph-cats-with-files
4182      (lex pos directory morphs-to-cats-prefix
4183           &key (thesaurus 'thesaurus-full)(level nil))
4184    (let* ((morphs-to-cats-file (format nil "~A~A.morphs.to.cats"  directory morphs-to-cats-prefix)))
4185      (if (null *morphs-to-cats*)
4186          (progn
4187            (format t "~%Loading morph-to-cats from file ~a." morphs-to-cats-file)
4188            (load-morphs-to-cats morphs-to-cats-file)))
4189      (get-morph-cats lex pos *morphs-to-cats* :thesaurus thesaurus :level level)))
4190
4191  ;;;Get the cat or cats for a given morph.
4192  (defun get-morph-cats (lex pos morphs-to-cats &key (thesaurus 'thesaurus-full)(level nil))
4193    (let* ((pos-morphrecs (third (assoc pos morphs-to-cats :test #'equal)))
4194           (morphrec (assoc lex pos-morphrecs :test #'equal))
4195           (thesaurus-entries (get-morphrec-element morphrec 'thesaurus-entries))
4196           output)
4197      (loop for entry in thesaurus-entries do
```

LINE # TEXT

```
4198            (let ((cat-codes
4199                    (get-thesaurus-entry-element entry 'cat-codes :thesaurus thesaurus)))
4200              (if cat-codes
4201                  (if (consp cat-codes)
4202                      (setq output (append output cat-codes))
4203                      (setq output (append output (list cat-codes)))))))
4204
4205      ;;Iff LEVEL is specified, collect ancestors for all associated codes
4206      ;;and then filter, using only codes at the right level.
4207      ;;Else any codes supplied by the thesaurus are used,
4208      ;;and these can be at several levels.
4209      (if level
4210          (let ((all-ancestors nil)
4211                (ancestors-at-right-level nil))
4212            (loop for cat-code in output do
4213                  (let ((this-cat-plus-ancestors
4214                         (adjoin cat-code (get-code-ancestors cat-code) :test #'equal)))
4215                    (loop for ancestor in this-cat-plus-ancestors do
4216                          (setq all-ancestors
4217                                (adjoin ancestor all-ancestors :test #'equal)))))
4218            (loop for ancestor in all-ancestors do
4219                  (if (equal (length ancestor) level)
4220                      (setq ancestors-at-right-level
4221                            (adjoin ancestor ancestors-at-right-level :test #'equal))))
4222            (setq output ancestors-at-right-level)))
4223      output))
4224
4225  ;;;============================================================================
4226  ;;;  II.G.6.  GETTING MORPH DICTIONARY FORMS
4227  ;;;============================================================================
4228
4229  (defun get-dictionary-form-for-verb-stem (verb-stem directory thesaurus-prefix)
4230    (let ((lexicon
4231           (read-from-file
4232            (format nil "~A~A.verb.dictionary.forms" directory thesaurus-prefix))))
4233      (if (null lexicon)
4234          (error "Null conversion lexicon retrieved from ~A~A.verb.dictionary.forms" directory thesaurus-prefix))
4235      (or (second (assoc verb-stem lexicon :test #'equal))
4236          verb-stem)))
4237
4238  (defun get-dictionary-form-for-common-noun (common-noun directory thesaurus-prefix)
4239    (let ((lexicon
4240           (read-from-file
4241            (format nil "~A~A.common.noun.dictionary.forms" directory thesaurus-prefix))))
4242      (if (null lexicon)
4243          (error "Null conversion lexicon retrieved from ~A~A.common.noun.dictionary.forms" directory thesaurus-prefix))
4244
4245      ;;;Note OR to handle failure, since not every noun is listed.
4246      (or (second (assoc common-noun lexicon :test #'equal))
4247          common-noun)))
4248
4249  ;;;============================================================================
4250  ;;;  II.G.7.  MAKING CATS-TO-MORPHS
4251  ;;;============================================================================
4252
4253  (defun make-cats-to-morphs-with-files (directory corpus-prefix &key (thesaurus 'thesaurus-full))
4254    (let* ((morphs-to-cats-file (format nil "~A~A.morphs.to.cats" directory corpus-prefix))
4255           (cats-to-morphs-file (format nil "~A~A.cats.to.morphs" directory corpus-prefix))
4256           (morphs-to-cats (read-from-file morphs-to-cats-file))
4257           (cats-to-morphs
4258            (make-cats-to-morphs morphs-to-cats '(root nil nil) :thesaurus thesaurus)))
4259      (write-to-file-no-pp cats-to-morphs cats-to-morphs-file)))
4260
4261  (defun make-cats-to-morphs (morphs-to-cats cats-to-morphs &key (thesaurus 'thesaurus-full))
4262    (loop for pos in morphs-to-cats do
4263          (let ((morphrecs-this-pos (third pos)))
4264            (loop for morphrec in morphrecs-this-pos do
4265                  (let ((thesaurus-entries (get-morphrec-element morphrec 'thesaurus-entries)))
4266                    (loop for entry in thesaurus-entries do
4267                          (let ((cat-codes
4268                                 (get-thesaurus-entry-element entry 'cat-codes :thesaurus thesaurus)))
4269                            (if cat-codes
4270                                (if (not (consp cat-codes))
4271                                    (setq cat-codes (list cat-codes))))
4272                            (loop for cat-code in cat-codes do
4273                                  (setq cats-to-morphs
4274                                        (insert-morphrec-into-cats-to-morphs
4275                                         morphrec cat-code cats-to-morphs)))))))))
4276    cats-to-morphs)
4277
4278  ;;;NON-DESTRUCTIVELY alters CATS-TO-MORPHS, initially '(root nil nil).
4279  (defun insert-morphrec-into-cats-to-morphs (morphrec code cats-to-morphs)
4280    (let* ((ancestor-codes-top-down (append (reverse (get-code-ancestors code)) (list code)))
4281           (current-node cats-to-morphs))
4282      (loop for object in ancestor-codes-top-down do
4283            (let ((lower-node
4284                   (assoc object (get-node-structure-element current-node 'daughters)
4285                          :test #'equal)))
4286              (if lower-node
4287                  (setq current-node lower-node)
4288                  (let* ((new-lower-node (make-node-structure object))
4289                         (new-current-node
4290                          ;;NON-DESTRUCTIVELY add new-lower-node to daughters of current-node.
4291                          (put-node-structure-element
4292                           current-node
4293                           'daughters
4294                           (cons new-lower-node
4295                                 (get-node-structure-element current-node 'daughters)))))
4296                    (setq cats-to-morphs
4297                          (subst new-current-node current-node cats-to-morphs))
4298                    (setq current-node new-lower-node)))))
4299
4300      (let ((new-current-node
4301             (add-node-structure-element current-node 'morphrecs morphrec)))
4302        (setq cats-to-morphs
4303              (subst new-current-node current-node cats-to-morphs)))
4304      cats-to-morphs))
4305
4306  (defun collect-morphs-without-entries (morphs-to-cats-global-file pos output-file)
4307    (let* ((global (read-from-file morphs-to-cats-global-file))
4308           (pos-record (assoc pos global :test #'equal))
4309           (morphrecs-without-entries
4310            (second (assoc 'morphrecs-without-entries (third pos-record) :test #'equal )))
4311           (morphs-without-entries
4312            (loop for morphrec in morphrecs-without-entries collect
4313                  (list (first morphrec)))))
4314      (write-to-file-no-pp morphs-without-entries output-file)))
4315
4316  ;;;============================================================================
4317  ;;;  II.G.8.  QUERYING CATS-TO-MORPHS
```

LINE # TEXT

```
4318 ;;;============================================================================
4319
4320 (defvar *CATS-TO-MORPHS* nil)
4321
4322 (defun load-cats-to-morphs
4323     (cats-to-morphs-file)
4324   (setq *CATS-TO-MORPHS*
4325         (read-from-file cats-to-morphs-file)))
4326
4327 (defun get-cat-morphs-with-files
4328     (cat directory morphs-to-cats-prefix &key (pos nil))
4329   (let* ((cats-to-morphs-file (format nil "~A~A.cats.to.morphs"  directory morphs-to-cats-prefix)))
4330     (if (null *cats-to-morphs*)
4331         (progn
4332           (format t "~%Loading cats-to-morphs from file ~a." cats-to-morphs-file)
4333           (load-cats-to-morphs cats-to-morphs-file)))
4334     (get-cat-morphs cat *cats-to-morphs* :pos pos)))
4335
4336 (defun get-cat-morphs (cat cats-to-morphs &key (pos nil))
4337   (let* ((cat-node (get-cats-to-morphs-node cat cats-to-morphs)))
4338     (if (null cat-node)
4339         (error "In get-cat-morphs, get-cats-to-morphs-node found no node for the specified cat."))
4340     (get-all-morphs-under-node cat-node :pos pos)))
4341
4342 ;;;Given a node in a cats-to-morphs tree structure,
4343 ;;;returns all morphrecs under that node.
4344 (defun get-all-morphs-under-node (current-node &key (pos nil))
4345   (let ((output nil)
4346         (daughters
4347          (get-node-structure-element
4348           current-node 'daughters)))
4349
4350     ;;This is a terminal.
4351     (if (null daughters)
4352         (let ((morphrecs
4353                (get-node-structure-element current-node 'morphrecs)))
4354           (if pos
4355               (let ((filtered-morphrecs
4356                      (loop for morphrec in morphrecs
4357                            when
4358                            (equal
4359                             pos
4360                             (get-morphrec-element morphrec 'pos))
4361                            collect
4362                            morphrec)))
4363                 (setq morphrecs filtered-morphrecs)))
4364           (setq output
4365                 (append morphrecs output)))
4366
4367         ;;Else recurse.
4368         (loop for daughter in daughters do
4369               (setq output
4370                     (append output (get-all-morphs-under-node daughter :pos pos)))))
4371     output))
4372
4373 ;;;Retrieves a node in the CATS-TO-MORPHS tree structure.
4374 (defun get-cats-to-morphs-node (cat cats-to-morphs)
4375   (let* ((ancestor-codes-top-down (append (reverse (get-code-ancestors cat)) (list cat)))
4376          (current-node cats-to-morphs)
4377          (top-code nil))
4378     (loop while ancestor-codes-top-down do
4379           (setq top-code (first ancestor-codes-top-down))
4380           (setq ancestor-codes-top-down
4381                 (cdr ancestor-codes-top-down))
4382           (let ((lower-node
4383                  (assoc top-code
4384                         (get-node-structure-element
4385                          current-node 'daughters)
4386                         :test #'equal)))
4387             (if lower-node
4388                 (setq current-node lower-node)
4389                 (error "In GET-CATS-TO-MORPHS-NODE, ASSOC failed to find a specified node on path down to cat."))))
4390     current-node))
4391
4392 ;;;****************************************************************************
4393 ;;;
4394 ;;;  III. KEE CONVERSION
4395 ;;;
4396 ;;;****************************************************************************
4397
4398 ;;;Plain KEE, without special J-KEE (Japanese KEE) facilities, will not handle Japanese characters.
4399 ;;;As a work-around to permit interfacing, we can convert specified CO-OC files to romaji.
4400 ;;;This function converts MORPHRECS-INDEXED to permit KEE load of morphrecs.
4401 ;;;See also separate file CO-OC-TO-KEE.LISP, and see documents.
4402 (defun convert-morphrecs-indexed-to-romaji-with-files (directory corpus-prefix)
4403   (let* ((morphrecs-indexed-file (format nil "~A~A.morphrecs.indexed"  directory corpus-prefix))
4404          (morphrecs-indexed-romaji-file (format nil "~A~A.morphrecs.indexed.romaji"  directory corpus-prefix))
4405          (morphrecs-indexed (read-from-file morphrecs-indexed-file))
4406          (output nil))
4407     (if (null '*kanji-to-romaji-mappings*)
4408         (setq '*kanji-to-romaji-mappings*
4409               (read-from-file (format nil "~A~A.kanji.to.romaji" directory corpus-prefix))))
4410     (loop for pos-record in morphrecs-indexed do
4411           (let* ((pos (first pos-record))
4412                  (morphs-in-pos (second pos-record))
4413                  (morphs-in-pos-romaji
4414                   (loop for morph-in-pos in morphs-in-pos collect
4415                         (get-romaji-for-kanji morph-in-pos)))
4416                  (morphrecs-in-pos (third pos-record))
4417                  (morphrecs-in-pos-romaji
4418                   (loop for morphrec-in-pos in morphrecs-in-pos collect
4419                         (let* ((lex (get-morphrec-element morphrec-in-pos 'lex))
4420                                (lex-romaji (get-romaji-for-kanji lex))
4421                                (morphrec-romaji
4422                                 (put-morphrec-element morphrec-in-pos 'lex lex-romaji)))
4423                           morphrec-romaji))))
4424             (setq output
4425                   (cons
4426                    (list pos morphs-in-pos-romaji morphrecs-in-pos-romaji)
4427                    output))))
4428     (setq output (reverse output))
4429     (write-to-file output morphrecs-indexed-romaji-file)))
4430
4431 (defvar *kanji-to-romaji-mappings* nil)
4432
4433 (defun get-romaji-for-kanji (kanji)
4434   (second (assoc kanji *kanji-to-romaji-mappings* :test #'equal)))
4435
4436
4437
```

LINE # TEXT

```
(in-package 'KEE)
;;;NOTE!!!Runs in KEE in Lucid Common Lisp 4.0, not 4.1.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;=================================================================================
;;;           CO-OC: Building Knowledge Bases of Co-occurrence Information
;;;                          AUX Functions for KEE Interface
;;;=================================================================================
;;;
;;;                                 Mark Seligman
;;;                                   Dec. 1994
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;CONTENTS

;;;I. BORROWED FUNCTIONS FROM CO-OC.LISP
;;;  A. READING/WRITING
;;;  B. DEFINING NECESSARY STRUCTURES
;;;    1. DEFINING MORPHREC DATASTRUCTURE
;;;    2.  DEFINING CATREC DATASTRUCTURE
;;;    3.  DEFINING NODE-STRUCTURE
;;;  C. OTHER

;;;II. FUNCTIONS SPECIFICALLY FOR KEE
;;;  A. HACKS
;;;  B. CREATING KEE UNITS
;;;    1.  KEE UNITS CORRESPONDING TO MORPHRECS
;;;    2.  KEE UNITS CORRESPONDING TO CATRECS

;;;*********************************************************************************
;;;
;;;  I. BORROWED FUNCTIONS FROM CO-OC.LISP
;;;
;;;*********************************************************************************

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; I.A. READING/WRITING
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;To avoid problems with pprint, etc.
(setq *print-level* 20)
(setq *print-shared* nil)
(setq *print-circle* nil)

;;;Reads one s-exp from file.
;;;If no such file, returns nil.
(defun read-from-file (file)
  (with-open-file
      (input-stream file :direction :input :if-does-not-exist nil)
    (if input-stream
        (read input-stream)
        nil)))

;;;Writes (pprints) one s-exp to file.
(defun write-to-file (output file)
  (with-open-file
      (output-stream
       file
       :direction :output
       :if-exists :new-version
       :if-does-not-exist :create)
    (pprint output output-stream)))

;;;Writes (but does not pprint) one s-exp to file.
(defun write-to-file-no-pp (output file)
  (with-open-file
      (output-stream file
        :direction :output
        :if-exists :new-version
        :if-does-not-exist :create)
    (format output-stream "~s" output)))

;;;Takes SERIES of s-exps from file and returns LIST of s-exps.
(defun read-loop (file)
  (let (output)
    (with-open-file
        (input-stream file :direction :input)
      (let ((new-value t))
        (loop while new-value do
              (setq new-value (read input-stream nil))
              (setq output (cons new-value output)))
        (reverse (cdr output))))))

;;;Takes LIST of s-exps and writes them into a file as SERIES, without parens.
;;;If non-nil numbers?, prints ;;;No.X before each element.
(defun write-loop (list file &optional numbers?)
  (with-open-file
      (output-stream
       file
       :direction :output
       :if-exists :new-version
       :if-does-not-exist :create)
    (let ((count 1))
      (loop for x in list do
            (if numbers?
                (progn
                  (format output-stream "~2%;;;No.~S~%" count)
                  (setq count (1+ count))))
            (print x output-stream)))))

;;;Special aux fn for co-oc
(defun get-global-value-from-file (file attribute)
  (let* ((all-values (read-loop file))
         (fetched (assoc attribute all-values :test #'equal)))
    (second fetched)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  I.B. DEFINING NECESSARY STRUCTURES
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;=================================================================================
;;;  I.B.1.  DEFINING MORPHREC DATASTRUCTURE
;;;=================================================================================

(defvar *morphrec-elements*
  '((lex 0)
```

LINE # TEXT

```
121      (pos 1)
122      (segments 2)
123      (hit-count 3)
124      (morph-segments-count 4)
125      (occur-in-segment-probability 5)
126      (occur-in-segment-information 6)
127      (occur-in-segment-probability-smoothed 7)
128      (morph-unigram-probability 8)
129      (morph-unigram-information 9)
130      (morph-unigram-probability-smoothed 10)
131      (thesaurus-entries 11)))          ;<<<
132
133  (defun make-morphrec (lex pos)
134    (let ((morphrec
135            (loop for element in *morphrec-elements* collect nil)))
136      (setq morphrec (put-morphrec-element morphrec 'lex lex))
137      (setq morphrec (put-morphrec-element morphrec 'pos pos))
138      morphrec))
139
140  (defun get-morphrec-element-position (element-name)
141    (second
142     (assoc element-name *morphrec-elements*)))
143
144  (defun get-morphrec-element (morphrec element-name)
145    (let ((position (get-morphrec-element-position element-name)))
146      (nth position morphrec)))
147
148  (defun put-morphrec-element (morphrec element-name value)
149    (let ((morphrec-copy (copy-list morphrec))
150          (position (get-morphrec-element-position element-name)))
151      (setf (nth position morphrec-copy) value)
152      morphrec-copy))
153
154  (defun add-morphrec-element (morphrec element-name value)
155    (let ((morphrec-copy (copy-list morphrec))
156          (old-value (get-morphrec-element morphrec element-name))
157          (new-value nil)
158          (position (get-morphrec-element-position element-name)))
159      (if (not (listp old-value))
160          (error "old value not a list")
161          (progn (setq new-value (adjoin value old-value :test #'equal))
162                 (setf (nth position morphrec-copy) new-value)))
163      morphrec-copy))
164
165  ;;;============================================================================
166  ;;;  I.B.2.  DEFINING CATREC DATASTRUCTURE
167  ;;;============================================================================
168
169  (defvar *catrec-elements*
170    '((code 0)
171      (segments 1)
172      (hit-count 2)
173      (cat-segments-count 3)
174      (occur-in-segment-probability 4)
175      (occur-in-segment-information 5)
176      (occur-in-segment-probability-smoothed 6)
177      (cat-unigram-probability 7)
178      (cat-unigram-information 8)
179      (cat-unigram-probability-smoothed 9)))
180
181  (defun make-catrec (code)
182    (let ((catrec
183            (loop for element in *catrec-elements* collect nil)))
184      (setq catrec (put-catrec-element catrec 'code code))
185      catrec))
186
187  ;;;<<first position is zero, not 1!!
188  ;;;Similar to morphrec, see above.
189  (defun get-catrec-element-position (element-name)
190    (second
191     (assoc element-name
192            *catrec-elements*)))
193
194  (defun get-catrec-element (catrec element-name)
195    (let ((position (get-catrec-element-position element-name)))
196      (nth position catrec)))
197
198  (defun put-catrec-element (catrec element-name value)
199    ;;;Using copy-list, not the more general copy-tree
200    (let ((catrec-copy (copy-list catrec))
201          (position (get-catrec-element-position element-name)))
202      (setf (nth position catrec-copy) value)
203      catrec-copy))
204
205  (defun add-catrec-element (catrec element-name value)
206    ;;;Using copy-list, not the more general copy-tree
207    (let ((catrec-copy (copy-list catrec))
208          (old-value (get-catrec-element catrec element-name))
209          (new-value nil)
210          (position (get-catrec-element-position element-name)))
211      (if (not (listp old-value))
212          (error "Old value not a list.")
213          (progn (setq new-value (adjoin value old-value :test #'equal))
214                 (setf (nth position catrec-copy) new-value)))
215      catrec-copy))
216
217  ;;;============================================================================
218  ;;;  I.B.3.  DEFINING NODE-STRUCTURE
219  ;;;============================================================================
220
221  (defvar *node-structure-elements*
222    '((object 0)
223      (daughters 1)
224      (catrec 2)
225      (morphrecs 3)))
226
227  (defun make-node-structure (object)
228    (let ((node-structure
229            (loop for element in *node-structure-elements* collect nil)))
230      (setq node-structure (put-node-structure-element node-structure 'object object))
231      node-structure))
232
233  (defun get-node-structure-element-position (element-name)
234    (second
235     (assoc element-name
236            *node-structure-elements*)))
237
238  (defun get-node-structure-element (node-structure element-name)
239    (let ((position (get-node-structure-element-position element-name)))
240      (nth position node-structure)))
```

LINE # TEXT

```
;;;Non-destructive: returns a copy of the input node-structure with
;;;the specified modification.
(defun put-node-structure-element (node-structure element-name value)
  (let ((node-structure-copy (copy-list node-structure))
        (position (get-node-structure-element-position element-name)))
    (setf (nth position node-structure-copy) value)
    node-structure-copy))

(defun add-node-structure-element (node-structure element-name value)
  (let ((node-structure-copy (copy-list node-structure))
        (old-value (get-node-structure-element node-structure element-name))
        (new-value nil)
        (position (get-node-structure-element-position element-name)))
    (if (not (listp old-value))
        (error "old value not a list")
        (progn (setq new-value (adjoin value old-value :test #'equal))
               (setf (nth position node-structure-copy) new-value)))
    node-structure-copy))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  I.C. OTHER
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun get-morphrecs-for-pos (pos morphrecs-indexed)
  (third (assoc pos morphrecs-indexed :test #'equal)))

;;;**********************************************************************************
;;;
;;;  II. FUNCTIONS SPECIFICALLY FOR KEE
;;;
;;;**********************************************************************************

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  II.A. HACKS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun rl () (load "~/survey/jap/topic/keefns.lisp"))

(defun run-demo ()
  (do-it)
  (do-it2))

(defun do-it ()
  (make-kee-units-for-morphrecs-with-files "~/survey/jap/data/" "1a.8b"))

(defun do-it2 ()
  (make-kee-units-for-catrecs-with-files "~/survey/jap/data/" "1a.4b"))

(defun no-morphs ()
  (delete.unit 'morphs nil t))

(defun no-cats ()
  (delete.unit 'cats nil t))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;  II.B. CREATING KEE UNITS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;==================================================================================
;;;  II.B.1.  KEE UNITS CORRESPONDING TO MORPHRECS
;;;==================================================================================

(defun make-kee-units-for-morphrecs-with-files (directory corpus-prefix &key (pos-of-interest '(common-noun verb)))
  (let* ((morphrecs-indexed-romaji-file (format nil "~A~A.morphrecs.indexed.romaji"  directory corpus-prefix))
         (morphrecs-indexed-romaji (read-from-file morphrecs-indexed-romaji-file)))

    ;;Make MORPHS class.
    (create.unit 'morphs 'co-oc.kb nil nil)

    ;;Make member slots for MORPHS class.
    (loop for element in *morphrec-elements* do
          (let ((slot-name (first element)))
            (create.slot 'morphs slot-name 'member)))

    ;;Make all pos class units and members (instances) for pos of interest.
    (loop for pos in pos-of-interest do
          (create.unit pos 'co-oc.kb 'morphs)

          (let ((morphrecs
                  (get-morphrecs-for-pos pos morphrecs-indexed-romaji)))
            (loop for morphrec in morphrecs do
                  (let* ((lex-romaji (get-morphrec-element morphrec 'lex)))

                    (create.unit lex-romaji 'co-oc.kb nil pos)
                    (loop for element in *morphrec-elements* do
                          (let* ((element-name (first element))
                                 (values (list (get-morphrec-element morphrec element-name))))
                            (put.values lex-romaji element-name values))))))))))

;;;==================================================================================
;;;  II.B.2.  KEE UNITS CORRESPONDING TO CATRECS
;;;==================================================================================

(defun make-kee-units-for-catrecs-with-files (directory corpus-prefix)
  (let* ((catrecs-indexed-file (format nil "~A~A.catrecs.indexed"  directory corpus-prefix))
         (catrecs-indexed (read-from-file catrecs-indexed-file)))

    ;;Make CATS class.
    (create.unit 'cats 'co-oc.kb nil nil)

    ;;Make member slots for CATS class.
    (loop for element in *catrec-elements* do
          (let ((slot-name (first element)))
            (create.slot 'cats slot-name 'member)))

    ;;Recursive descent thru catrecs-indexed.
    ;;Make cat units as we go.
    (make-cat-units-for-all-daughter-nodes catrecs-indexed 'cats)))

;;;CURRENT-NODE at top-level call is CATRECS-INDEXED.
;;;PARENT-UNIT at top level call is CATS.
(defun make-cat-units-for-all-daughter-nodes (current-node parent-unit)
  (let ((daughter-nodes
         (get-node-structure-element
          current-node 'daughters)))

    ;;If this is a terminal cat, without daughters, do nothing.
    (if (null daughter-nodes)
        nil
```

```
;;Else there are daughters.
(loop for daughter-node in daughter-nodes do
      ;;Make a daughter unit in KEE representing the daughter node.
      (let* ((code-string (string-upcase (get-catrec-element daughter-node 'code)))
             (daughter-unit-name (intern (format nil "~A.CAT" code-string)))
             (daughter-unit
              (create.unit daughter-unit-name 'co-oc.kb parent-unit nil)))

        ;;Transfer slot values from DAUGHTER-NODE to DAUGHTER-UNIT.
        (loop for element in *catrec-elements* do
              (let* ((element-name (first element))
                     (catrec (get-node-structure-element daughter-node 'catrec))
                     (values (list (get-catrec-element catrec element-name))))
                (put.values daughter-unit element-name values)))

        ;;Then recurse.
        (make-cat-units-for-all-daughter-nodes daughter-node daughter-unit))))))
```