Function tagging

Don Blaheta

September 24, 2002

Contents

Abstract

Parsing sentences using statistical information gathered from a treebank was first examined more than a decade ago and is by now a fairly well-studied problem; it has been attacked with a wide variety of tools, including decision trees, probabilistic context-free grammars, maximum entropy, and data-oriented parsing. Nearly all this work, however, has focused entirely on low-level syntactic structure: a bracketing (or dependency labelling) with simple constituent labels like NP, VP, or SBAR. The Penn Treebank contains a great deal of additional syntactic and semantic information from which to gather statistics; reproducing more of this information automatically is a goal which has so far been mostly ignored. I am interested in making it possible to recover some of this information—the function tags—automatically.

In this work I will present a system that utilises a maximum-entropy-inspired algorithmic framework along with a number of commonly used features (label, syntactic head, etc) to predict function tags with relatively high accuracy. I will then present two other algorithmic frameworks and a number of new features to be used with them. I propose to use these expanded systems to improve performance on the function tagging task, and having done so, analyse the results to determine which features were most helpful in the task as a whole and in its various subtasks.

1 Introduction

Parsing sentences using statistical information gathered from a treebank was first examined more than a decade ago in (Chitrao and Grishman, 1990) and is by now a fairly well-studied problem; it has been attacked with a wide variety of tools, from decision trees (Magerman, 1995), to probabilistic context-free grammar (Charniak, 1997), to maximum entropy (Ratnaparkhi, 1999; Charniak, 2000), to data-oriented parsing (Bod, 2001). Nearly all evaluate themselves by how well they recovered the parse structure and constituent labelling of the hand-parsed treebank; some evaluate the dependency structure (Collins, 1999; Hockenmaier and Steedman, 2002). Recently there has been a revival of interest in the old NLP goal of language modelling, and several researchers have also been reporting perplexity scores (Chelba, 2000; Roark, 2001; Charniak, 2001).

Through all that research, however, the parsing process (and its evaluation) has focused entirely on low-level syntactic structure, a bracketing (or dependency labelling) with simple constituent labels like NP, VP, or SBAR. The Penn Treebank contains a great deal of additional syntactic and semantic information from which to gather statistics; reproducing more of this information automatically is a goal which has so far been mostly ignored. We are interested in making it possible to recover some of this information—the function tags—automatically.

At a high level, we can simply say that having syntactic and semantic tag information for a given text is useful just because any further information would help. Syntactic tags are useful for any application trying to follow the thread of the text—they find the 'who does what' of each clause, which can be useful to gain information about the situation or to learn more about the behaviour of the words in the sentence. For instance, any algorithm that needs to know the subject of a sentence would benefit from actually having that subject, rather than relying on an easy but stupid algorithm like "first noun phrase" or "verb phrase's left sibling". Noting a topicalised constituent could help in discourse analysis, or pronoun resolution. Semantic tags help to find those constituents behaving in ways not conforming to their labelled type (e.g. noun phrases acting adverbially), as well as more specifically identifying the role of adjunct prepositional and adverbial phrases. Information retrieval applications specialising in describing events, as with a number of the MUC applications, could greatly benefit from some of these in determining the where-when-why of things.

2 Related work

To our knowledge, there has been no attempt so far to recover the Penn treebank function tags, as such, during or after parsing text. There have, however, been a number of projects that recovered similar information. We will analyse how they compare to this work in Section 9.

2.1 Collins

Collins (1997) approaches the problem of distinguishing adjuncts from complements. The motivation in that case was to improve parser performance—by guessing complement status during the parse, the statistics were a bit cleaner. That paper defines constituents as complements or adjuncts based on a combination of label and function tag information. This boolean condition is then used to train the improved parser.

The system uses a generative model to evaluate parse quality, and the complement information is used as follows. After generating the head-containing child of a constituent (conditioned on the constituent and its head word), left and right subcat frames are chosen conditional on that head-containing child (and the previously-used conditioning info). A subcat frame is simply a bag of labels that are subcategorised by, i.e. complement to, the parent. Given the subcat frame, then, and all the previous conditioning information, the actual labels of the other children are generated. Collins does not report his results on the complement tagging, but reports that using the complement information improves his parser's accuracy by 0.6% (0.7% in labelled recall, from 87.4% to 88.1%, and 0.5% in labelled precision, from 88.1% to 88.6%).

2.2 Gildea and Jurafsky

Gildea and Jurafsky (2000) present work on automatic marking of semantic roles. Their work is within the FrameNet framework: within *domains* such as "communication" or "cognition", there exist *frames* such as "conversation", "statement", and "judgement". Each frame then contains a number of words (of various parts of speech) that are semantically related (e.g. "argue", "debate", "discussion", and "tiff" in the "conversation" frame), and all those words share the same frame elements (e.g. PROTAGONIST, TOPIC, and MEDIUM, again for the "conversation" frame). The task of the labelling system, then, was to correctly mark sentence constituents that instantiated frame elements of some target predicate.

The system used empirically observed probabilities, conditioned on several things: phrase label, syntactic role, head, position, and target predicate. The backoff model used was a lattice: a less-specific distribution would be used "only when no data was present for any more specific distribution". To test the system, they took a portion of FrameNet, threw out the frame element labels, and measured the percentage of times the system assigned the correct labels; the best version (with lattice backoff and head clustering) performed at 81.2%. Note that the bracketing, i.e. where in the sentence there *were* frame elements, remained in the input to the system; a separate experiment reported in the same paper on identifying unlabelled frame elements showed performance of 66% (with an additional 15% partial matches). Performance of the main labelling system on the correctly identified frame elements was 79.6%, comparable to the hand-bracketed input.

2.3 Brants, Skut, and Krenn

Brants et al. (1997) attack the problem of building a German-language treebank at various levels of automation. Like the Penn WSJ treebank project, Brants et al. begin with POS-tagged newspaper text; the annotation consists of phrase structure labels like NP and S, and grammatical function labels like SB 'subject' and HD 'head'. This last category is similar to what we will call the grammatical tags of the Penn treebank, though there are differences as well. The main difference is that in the German project, *every* constituent has a grammatical function tag, so there are a number of tags for roles that are unmarked in the Penn treebank.

Their algorithm relies to some extent on this finer granularity of grammatical tags: they use an order-2 Markov model whose states correspond to the tags themselves. Each phrasal category has its own model for use when it is the parent node, and the emitted values are the phrasal category labels of the children. Their performance at the task was 94.2%, with substantial variation according to the parent category: children of S nodes were only tagged at 89.1% accuracy, while PP parents enjoyed an accuracy of 97.9%.

3 Function tags

In the Penn treebank, there are 20 tags (see Figure 1) that can be appended to constituent labels in order to indicate additional information about the syntactic or semantic role of the constituent. A full description and examples of each tag

ADV	Non-specific adverbial	MNR	Manner
BNF	Benefactive	NOM	Nominal
CLF	It-cleft	PRD	Predicate
CLR	'Closely related'	PRP	Purpose
DIR	Direction	PUT	Locative complement of 'put'
DTV	Dative	SBJ	Subject
EXT	Extent	TMP	Temporal
HLN	Headline	TPC	Topic
LGS	Logical subject	TTL	Title
LOC	Location	VOC	Vocative

Figure 1: Penn treebank function tags

can be found in the annotation guidelines (Bies et al., 1995); a short description can be found in Appendix A. We have divided them into four categories based on those in the bracketing guidelines; these can be seen in Figure 2. Also in Figure 2 are the total counts of each tag and category within sections 02–21 of the treebank (the training) and section 24 of the treebank (the development corpus), and both together. Finally, we give the percentage of all constituents tagged with any tag from a given category that are tagged with each specific tag ("**of cat%**"), and the percentage of all nonterminal constituents in the relevant sections that are tagged with each tag and category ("**total%**"). A constituent can be tagged with multiple tags, but never with two tags from the same category.[1] In actuality, the case where a constituent has tags from all four categories never happens, but constituents with three tags do occur (rarely).

3.1 The curious case of the CLR tag

One of the function tags to be found in the Penn treebank is CLR, which stands for "Closely Related". The bracketing guidelines say that this tag "marks constituents that occupy some middle ground between argument and adjunct of the verb phrase." Their main role seems to be in marking different varieties of phrasal verb, such as 'rely on' or 'put up with'. This is an interesting linguistic phenomenon, and useful to mark.

Unfortunately, it is not exactly a binary phenomenon; the degree to which a verb and some other constituent are phrasal, or "closely related", often lies very much in the eye of the beholder. Even a casual perusal of the development corpus will reveal a few CLR constituents that seem like they shouldn't be, and a large number that aren't but seem like they should be. Every reader, furthermore, will put different constituents in these two sets.

In earlier iterations of this work, we put the CLR tag into the fourth, "miscellaneous" category of function tags. At some 603 such tags in the development

[1] There is a single exception in the corpus: one constituent is tagged with -LOC-MNR. This appears to be an error.

	§§02–21	§24	total	of cat%	total%
All nonterminals	781665	27075	808740		
Grammatical	87814	3066	90880	100.00%	11.24%
DTV	423	15	438	0.48%	0.05%
LGS	2604	98	2702	2.97%	0.33%
PRD	15629	606	16235	17.86%	2.01%
PUT	224	11	235	0.25%	0.03%
SBJ	68912	2333	71245	78.39%	8.81%
VOC	22	3	25	0.02%	0.00%
Form/function	60995	2050	63045	100.00%	7.80%
ADV	6997	206	7203	11.42%	0.89%
BNF	44	1	45	0.07%	0.01%
DIR	5052	123	5175	8.20%	0.64%
EXT	1968	52	2020	3.20%	0.25%
LOC	15426	584	16010	25.39%	1.98%
MNR	3788	110	3898	6.18%	0.48%
NOM	4139	131	4270	6.61%	0.53%
PRP	3216	105	3321	5.26%	0.41%
TMP	20365	738	21103	33.47%	2.61%
Topicalisation (TPC)	3722	119	3841	100.00%	0.47%
Miscellaneous	966	35	1001	100.00%	0.12%
CLF	53	1	54	5.39%	0.01%
HLN	417	11	428	42.75%	0.05%
TTL	496	23	519	51.84%	0.06%

Figure 2: Categories of function tags and their relative frequencies

corpus, it was by far the dominant member of that category. A great deal of effort was spent on improving our performance in the category, but recently we discovered that (though this directive never made it into the documentation) users of the treebank were "supposed to strip out the CLR tag."[2] It seems the annotators were given a great deal of leeway and very little guidance as to how to use the tag, and much of the explanatory text in the bracketing guidelines was written after the fact to (roughly) fit how the annotators used the tag.

We have since stopped training, testing, or reporting results on the CLR tag, and restricted the fourth category to its other three, much rarer, members.

[2]Mitch Marcus, p.c., 7 July 2002

4 Features

We have found it useful to define our statistical model in terms of *features*. A 'feature', in this context, is a boolean-valued function, generally over parse tree nodes and either node labels or lexical items. Features can be fairly simple and easily read off the tree (e.g. 'this node's label is X', 'this node's parent's label is Y'), or slightly more complex ('this node's head's part-of-speech is Z'). This is concordant with the usage in the maximum entropy literature (Berger et al., 1996).

When using a number of known features to guess an unknown one, the usual procedure is to calculate the value of each feature, and then essentially look up the empirically most probable value for the feature to be guessed based on those known values. Due to sparse data, some of the features later in the list may need to be ignored; thus the probability of an unknown feature value would be estimated as

$$\begin{aligned} P(f|f_1, f_2, \ldots, f_n) \approx \\ \hat{P}(f|f_1, f_2, \ldots, f_j), \quad j \leq n \ , \end{aligned} \tag{1}$$

where $\hat{P}$ refers to an empirically observed probability. Of course, if features 1 through i only co-occur a few times in the training, this value may not be reliable, so the empirical probability is usually smoothed:

$$\begin{aligned} P(f|f_1, f_2, \ldots, f_i) \approx \\ & \lambda_i \hat{P}(f|f_1, f_2, \ldots, f_i) \\ + \quad & (1 - \lambda_i) P(f|f_1, f_2, \ldots, f_{i-1}) \ . \end{aligned} \tag{2}$$

The values for λ_i can then be determined according to the number of occurrences of features 1 through i together in the training.

One way to think about Equation 1 (and specifically, the notion that j will depend on the values of $f_1 \ldots f_n$) is as follows: We begin with the prior probability of f. If we have data indicating $\hat{P}(f|f_1)$, we multiply in that likelihood, while dividing out the original prior. If we have data for $\hat{P}(f|f_1, f_2)$, we multiply that in while dividing out the $\hat{P}(f|f_1)$ term. This is repeated for each piece of feature data we have; at each point, we are *adjusting* the probability we already have estimated. If knowledge about feature f_i makes f more likely than with just $f_1 \ldots f_{i-1}$, the term where f_i is added will be greater than one and the running probability will be adjusted upward. This gives us the new probability shown in Equation 3, which is exactly equivalent to Equation 1 since everything except the last numerator cancels out of the equation. The value of j is chosen such that features $f_1 \ldots f_j$ are sufficiently represented in the training data; sometimes all n features are used, but often that would cause sparse data problems. Smoothing is performed on this equation exactly as before: each term is interpolated between the empirical value and the prior estimated probability, according to a value of λ_i that estimates confidence. But aside from perhaps providing a new way to think about the problem, Equation 3 is not particularly useful as it is—it is exactly the same as what we had before. Its real usefulness

$$\begin{aligned} P(f|f_1, f_2, \ldots, f_n) &\approx \hat{P}(f)\frac{\hat{P}(f|f_1)}{\hat{P}(f)}\frac{\hat{P}(f|f_1,f_2)}{\hat{P}(f|f_1)}\cdots\frac{\hat{P}(f|f_1,f_2,\ldots,f_j)}{\hat{P}(f|f_1,f_2,\ldots,f_{j-1})}, \quad j \leq n \\ &\approx \prod_{i=0}^{j}\frac{\hat{P}(f|f_1,\ldots,f_{i-1},f_i)}{\hat{P}(f|f_1,\ldots,f_{i-1})} \end{aligned} \tag{3}$$

comes, as shown in (Charniak, 1999), when we move from the notion of a feature *chain* to a feature *tree*.

These feature chains don't capture everything we'd like them to. If there are two independent features that are each relatively sparse but occasionally carry a lot of information, then putting one before the other in a chain will effectively block the second from having any effect, since its information is (uselessly) conditioned on the first one, whose sparseness will completely dilute any gain. What we'd really like is to be able to have a feature *tree*, whereby we can condition those two sparse features independently on one common predecessor feature. As we said before, equation 3 represents, for each feature f_i, the probability of f based on f_i and all its predecessors, divided by the probability of f based only on the predecessors. In the chain case, this means that the denominator is conditioned on every feature from 1 to $i-1$, but if we use a feature tree, it is conditioned only on those features along the path to the root of the tree.

A notable issue with feature trees as opposed to feature chains is that the terms do *not* all cancel out. Every leaf on the tree will be represented in the

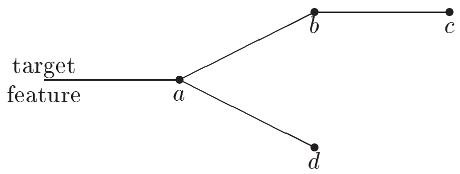

Figure 3: A small example feature tree

numerator, and every fork in the tree (from which multiple nodes depend) will be represented at least once in the denominator. For example: in Figure 3 we have a small feature tree that has one target feature and four conditioning features. Features b and d are independent of each other, but each depends on a; c depends directly only on b. The unsmoothed version of the corresponding equation would be

$$P(f|a,b,c,d) \approx \hat{P}(f)\frac{\hat{P}(f|a)}{\hat{P}(f)}\frac{\hat{P}(f|a,b)}{\hat{P}(f|a)}\frac{\hat{P}(f|a,b,c)}{\hat{P}(f|a,b)}\frac{\hat{P}(f|a,d)}{\hat{P}(f|a)},$$

(where P is the probability we are trying to derive, and $\hat{P}$ is the empirically observed probability); after cancelling of terms and smoothing, this gives us

$$P(f|a,b,c,d) \approx \frac{P(f|a,b,c)P(f|a,d)}{P(f|a)} . \tag{4}$$

Note that strictly speaking the result is not a probability distribution. It could be made into one with an appropriate normalisation—the so-called partition function in the maximum-entropy literature. However, if the independence assumptions made in the derivation of Equation 4 are good ones, the partition function will be close to 1.0. We assume this to be the case for our feature trees.

Now we return the discussion to function tagging. There are a number of features that seem to condition strongly for one function tag or another; we have assembled them into the feature tree shown in Figure 4.[3] This figure should be

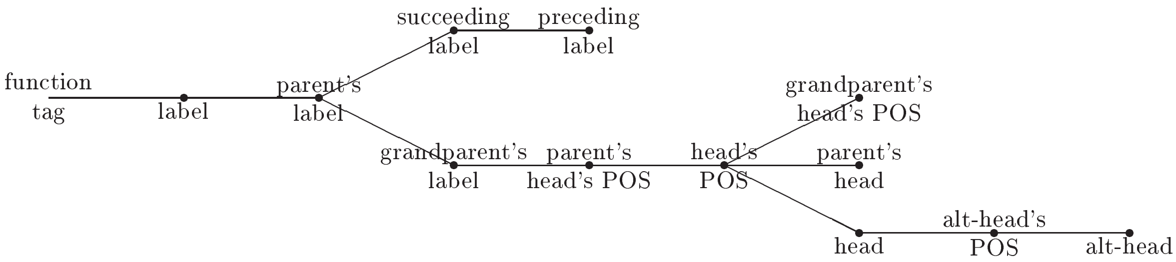

Figure 4: The feature tree used to guess function tags

relatively self-explanatory, except for the notion of an 'alternate head'; currently, an alternate head is only defined for prepositional phrases, and is the second child of the PP node. This data is very important in distinguishing, for example, 'in Budapest' (which is locative) from 'in April' (which is temporal).[4]

5 Experiment

In the training phase of our experiment, we gathered statistics on the occurrence of function tags in sections 2-21 of the Penn treebank. Specifically, for every

[3]The reader will note that the 'features' listed in the tree are in fact not boolean-valued; each node in the given tree can be assumed to stand for a chain of boolean features, one per potential value at that node, exactly one of which will be true.

[4]The word 'by' seems to be the most ambiguous preposition in this respect; it can give rise to five different function tags in contexts that differ only by the prepositional object:

The volume was turned up by John (LGS)
by 30 dB (EXT)
by the DJ table (LOC)
by a twist of the knob (MNR)
by 11pm (TMP)

constituent in the treebank, we recorded the presence of its function tags (or lack thereof) along with its conditioning information. From this we calculated the empirical probabilities of each function tag referenced in section 4 of this paper. Values of λ were determined using EM on the development corpus (treebank section 24).

Rather than test our ability to apply function tags to the hand-parsed text of the treebank, we chose to apply the function tags to a parse generated by a state of the art statistical parser. This, we feel, better models the sort of real-world use of the technology that we eventually envision. We discuss the extent to which this affects the results elsewhere in this paper (in Sections 7.4 and 8.1).

To test, then, we simply took the output of our parser on sentences of length 40 or less in the test corpus (treebank section 23), and applied a postprocessing step to add function tags. For each constituent in the tree, we calculated the likelihood of each function tag according to the feature tree in Figure 4, and for each category (see Figure 2) we assigned the most likely function tag (which might be the null tag).

6 Evaluation

To evaluate our results, we first need to determine what is 'correct'. The definition we chose is to call a constituent correct if there exists in the correct parse a constituent with the same start and end points, label, and function tag (or lack thereof). Since we treated each of the four function tag categories as a separate feature for the purpose of tagging, evaluation was also done on a per-category basis.

The denominator of the accuracy measure should be the maximum possible number we could get correct. In this case, that means excluding those constituents that were already wrong in the parser output; the parser we used attains 89% labelled precision-recall, so roughly 11% of the constituents are excluded from the function tag accuracy evaluation. (For reference, we have also included the performance of our function tagger directly on treebank parses; the slight gain that resulted is discussed below.)

Another consideration is whether to count non-tagged constituents in our evaluation. On the one hand, we could count as correct any constituent with the correct tag as well as any correctly non-tagged constituent, and use as our denominator the number of all correctly-labelled constituents. (We will henceforth refer to this as the 'with-null' measure.) On the other hand, we could just count constituents with the correct tag, and use as our denominators the total number of *tagged*, correctly-labelled constituents. We believe the latter number ('no-null') to be a better performance metric, as it is not overwhelmed by the large number of untagged constituents. Both are reported below.

	Baseline 1 (never tag)	Baseline 2 (always choose most likely tag) Tag	Precision	Recall	F-measure
Grammatical	87.397%	SBJ	10.200%	80.930%	18.116%
Form/Function	91.762%	TMP	2.977%	36.137%	5.501%
Topicalisation	99.393%	TPC	0.607%	100.00%	1.206%
Miscellaneous	99.734%	TTL	0.119%	44.762%	0.238%
Overall	94.571%	—	3.476%	64.025%	6.593%

Table 1: Baseline performance

	With-null Accuracy	———No-null——— Precision	Recall	F-measure
Grammatical	98.820%	94.803%	95.509%	95.155%
Form/Function	97.069%	80.458%	79.169%	79.148%
Topicalisation	99.924%	93.724%	93.724%	93.724%
Miscellaneous	99.782%	75.676%	26.667%	39.437%
Overall	98.899%	87.927%	88.631%	88.626%

Table 2: Performance within each category

7 Results

7.1 Baselines

There are, it seems, two reasonable baselines for this and future work. First of all, most constituents in the corpus have no tags at all, so obviously one baseline is to simply guess no tag for any constituent. Even for the most common type of function tag (grammatical), this method performs with 87% accuracy. Thus the with-null accuracy of a function tagger needs to be very high to be significant here.

The second baseline might be useful in examining the no-null accuracy values (particularly the recall): always guess the most common tag in a category. This means that every constituent gets labelled with -SBJ-TMP-TPC-TTL (meaning that it is a topicalised temporal subject that is also the title of some work). This combination of tags would never actually occur in real use, but is adequate for a baseline. The precision is, of course, abysmal, for the same reasons the first baseline did so well; but the recall is (as one might expect) substantial. The performances of the two baseline measures are given in Table 1.

7.2 Performance in individual categories

In Table 2, we give the results for each category using the feature tree shown in Figure 4.[5] The first column is the with-null accuracy, and the precision and recall values given are the no-null accuracy, as noted in Section 6.

Grammatical tagging performs the best of the four categories. Even using the more difficult no-null accuracy measure, it has a 95.7% accuracy. This seems to reflect the fact that grammatical relations can often be guessed based on constituent labels, parts of speech, and high-frequency lexical items, largely avoiding sparse-data problems. Topicalisation can similarly be guessed largely on high-frequency information, and performed almost as well (92.9%).

On the other hand, we have the form/function tags and the 'miscellaneous' tags. These are characterised by much more semantic information, and the relationships between lexical items are very important, making sparse data a real problem. All the same, it should be noted that the performance is still far better than the baselines. At 78.9%, we are getting nearly four-fifths of the form/function tags correct, which should be enough to at least be helpful; the miscellaneous category, dominated by the CLR tag, did less well, barely clearing 60%.

The "overall" performance that we report in the last row of Table 2 is calculated by summing the total number correct or wrong in each category, and then doing the same division used for the individual category scores. While this does result in a simple mean average for the "Accuracy" column (since every constituent is considered), the overall score in other columns is weighted toward the more common categories.

7.3 Performance with other feature trees

The feature tree given in Figure 4 is by no means the only feature tree we could have used. It was merely one of the first, chosen on an ad hoc basis by what seemed reasonable at the time, when we first began the research. Of twelve or so candidate trees, it was selected as the one that performed best on the development corpus. Later, we tried a number of different feature trees, and the results are reported below.

7.3.1 Searching for the optimal tree

The exploration of the feature tree search space was relatively thorough. We started with a feature tree of zero nodes (i.e. just the prior), and repeatedly perturbed it by either adding a new feature at the end of one branch, swapping two features in the middle, or moving one of the features at the end of one branch onto a different (possibly new) branch of the tree. At all points we kept the

[5] Though the feature tree is the same, the results in Table 2 are different from those reported in (Blaheta and Charniak, 2000). This is due to two factors: first, we moved to tagging all sentences instead of just those with no more than forty words, as the original reasons for doing so were no longer valid. Second, as noted in Section 3.1, we are now ignoring the CLR tag. Further discussion of the resulting performance hit can be found in Section 7.3.2.

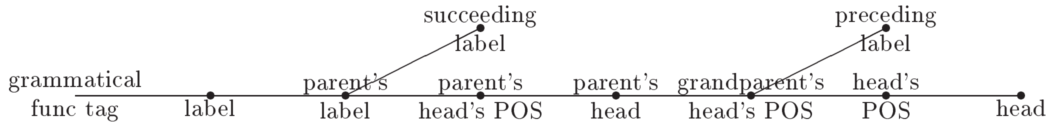

Figure 5: The feature tree used to guess grammatical tags

Figure 6: The feature tree used to guess form/function tags

several feature trees that performed best for each category on the development set (section 24 of the Penn treebank).

Since the original intent of this was primarily exploratory, the process was never fully automated, so the choice of "the several best" was to some extent subjective, and sometimes one better tree would be thrown out in favour of another, if the better tree was very similar both in form and in performance to an even better tree. In this way we hoped to avoid getting stuck in local maxima. Another factor in avoiding local maxima was that all categories were tested for each feature tree; this had the effect of retaining a number of fairly (increasingly) diverse trees in the pool for each category.

7.3.2 Results

The end result of this search gave trees that can thus achieve slight (one to three point) gains in each category. Figures 5, 6, 7, and 8 show these trees.

Table 3 gives the results achieved with these new trees. The first three categories offer noticeable improvement over that reported in Table 2. The grammatical tags increased by a bit over seven tenths of a percent, but this

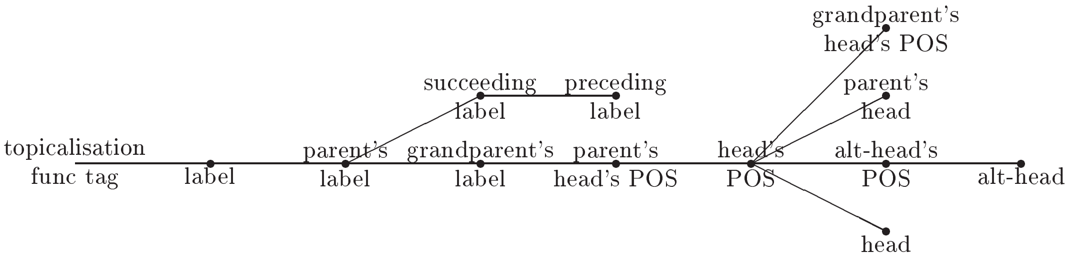

Figure 7: The feature tree used to guess topicalisation tags

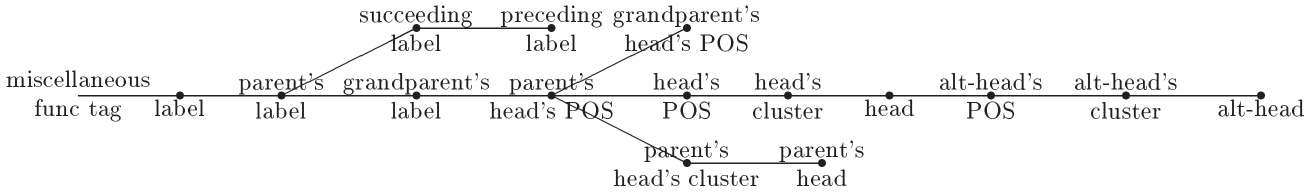

Figure 8: The feature tree used to guess miscellaneous tags

	With-null	———No-null———		
	Accuracy	Precision	Recall	F-measure
Grammatical	99.025%	96.455%	95.328%	95.888%
Form/Function	97.594%	86.693%	80.283%	83.365%
Topicalisation	99.944%	95.397%	95.397%	95.397%
Miscellaneous	99.759%	63.889%	21.905%	32.624%
Overall	99.081%	92.698%	88.721%	90.666%

Table 3: Performance with separate trees for each category

represents a 15% reduction in error. The form/function tags improved by 4.2% (20% error reduction); and the topicalisation tags improved by 1.7% (27% error reduction).

The performance on miscellaneous tags is not really indicative of anything other than performance on those two specific trees, since the optimisation was performed while CLR tags were still part of the equation. Apparently, the tree that performs best for CLR actually performs substantially worse on titles, headlines, and cleft constructions (the other members of the fourth category). It is perhaps worth noting that the performance of these trees on the development corpus shows a similar decrease: 42% on the Figure 4 tree, 34% on the Figure 8 tree.

	With-null	———No-null———		
	Accuracy	Precision	Recall	F-measure
Machine parsed	99.025%	92.698%	88.721%	90.666%
Hand parsed (treebank)	99.204%	93.952%	89.783%	91.820%

Table 4: Overall performance on different inputs

7.4 Performance on hand-parsed (treebank) text

The performance given in the first row of Table 4 is (like all previously given performance values) the function-tagger's performance on the correctly-labelled constituents output by our parser. For comparison, we also give its performance when run directly on the original treebank parse; since the parser's accuracy is about 89%, working directly with the treebank means roughly 12% more constituents are evaluated. This second version does slightly better.

8 Analysis

To analyse the errors our system was making, we first printed out every constituent with a function tag error. We then examined the sentence in which each occurred, and determined whether the error was in the algorithm or in the treebank, or elsewhere, as reported in Table 5. Of the errors we examined, less than half were due solely to an algorithmic failure in the function tagger itself. The next largest category was parse error: this function tagging algorithm requires parsed input, and in these cases, that input was incorrect and led the function tagger astray; had the tagger received the treebank parse, it would have given correct output. In just under a fifth of the reported "errors", the algorithm was correct and the treebank was definitely wrong. The remainder of cases we have identified either as Type C errors—wherein the tagger agreed with many training examples, but the "correct" tag agreed with many others (see Section 8.2.3)—or at least "dubious", in the cases that weren't common enough to be systematic inconsistencies but where the guidelines did not clearly prefer the treebank tag over the tagger output, or vice versa.

8.1 Parse error

It may at first seem strange that so many reported errors would be due to parse error, since (as stated in Section 6) we only count correctly parsed constituents in our evaluation. However, although the constituent itself may be correctly bracketed and labelled, its exterior conditioning information can still be incorrect. An example of this that actually occurred in the development corpus (section 24 of the treebank) is the 'that' clause in the phrase 'can swallow the premise that the rewards for such ineptitude are six-figure salaries', correctly

Algorithm error	44%
Parse error	20%
Treebank error	18%
Type C error	13%
Dubious	6%

Table 5: Analysis of reported errors

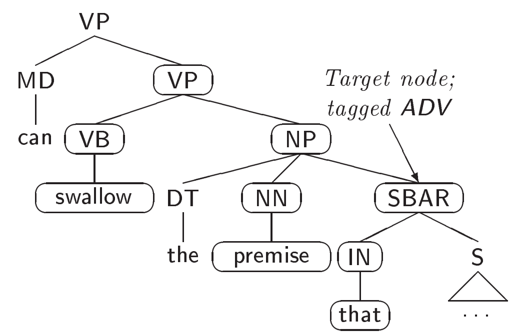

Figure 9: SBAR and conditioning info

diagrammed in Figure 9. The function tagger gave this SBAR an ADV tag, indicating an unspecified adverbial function. This seems extremely odd, given that its conditioning information (nodes circled in the figure) clearly show that it is part of an NP, and hence probably modifies the preceding NN. Indeed, the statistics give the probability of an ADV tag in this conditioning environment as vanishingly small.

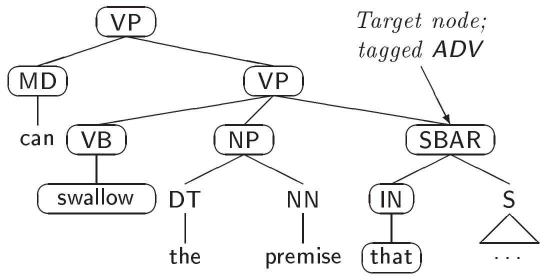

Figure 10: SBAR and conditioning info, as parsed

However, this was not the conditioning information that the tagger received. The parser had instead decided on the (incorrect) parse in Figure 10. As such, the tagger's decision makes much more sense, since an SBAR under two VPs whose heads are VB and MD is indeed rather likely to be an ADV. (For instance, the 'although' clause of the sentence 'he can help, although he doesn't want to.' has exactly the conditioning environment given in Figure 10, except that its predecessor is a comma; and such an SBAR would be correctly tagged ADV.) The SBAR itself is correctly bracketed and labelled, so it still gets counted in the statistics. In fact, since the only requirement for an individual constituent to be "correct" is that it have the correct label and subtend the correct portion of the sentence, every single other conditioning event could (in theory) be wrong! Actual occurrences aren't so extreme, but are sufficiently frequent that they are a significant source of error.

8.2 Treebank error

Another thing that lowers the overall performance somewhat is the existence of error and inconsistency in the treebank tagging. Performed by humans, the annotation inevitably has errors in it. We have designed a taxonomy for cataloguing corpus errors and some guidelines for correcting them, first published in (Blaheta, 2002).

8.2.1 Type A: Detectable errors

The easiest errors, which we have dubbed "Type A", are those that can be automatically detected and fixed. These typically come up when there would be multiple reasonable ways of tagging a certain interesting situation: the markup guidelines arbitrarily choose one, and the human annotator unthinkingly uses the other.

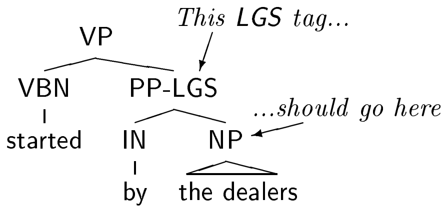

Figure 11: A function tag error of Type A

The canonical example of this sort of thing is the treebank's LGS tag, representing the "logical subject" of a passive construction. It makes a great deal of sense to put this tag on the NP object of the 'by' construction; it makes almost as much sense to tag the PP itself, especially since (given a choice) most other function tags are put there. The treebank guidelines specifically choose the former: "It attaches to the NP object of *by* and not to the PP node itself." (Bies et al., 1995) Nevertheless, in several cases the annotators put the tag on the PP, as shown in Figure 11. We can automatically correct this error by algorithmically removing the LGS tag from any such PP and adding it to the object thereof.

The unifying feature of all Type A errors is that the annotator's *intent* is still clear. In the LGS case, the annotator managed to clearly indicate the presence of a passive construction and its logical subject. Since the transformation from what was marked to what ought to have been marked is straightforward and algorithmic, we can easily apply this correction to all data.

8.2.2 Type B: Fixable errors

Next, we come to the Type B errors, those which are fixable but require human intervention at some point in the process. In theory, this category could include

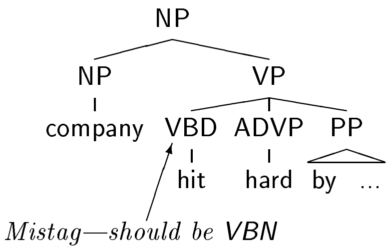

Figure 12: A part-of-speech error of Type B_1

errors that could be found automatically but require a human to fix; this doesn't happen in practice, because if an error is sufficiently systematic that an algorithm can detect it and be certain that it is in fact an error, it can usually be corrected with certainty as well. In practice, the instances of this class of error are all cases where the computer can't detect the error for certain. However, for all Type B errors, once detected, the correction that needs to be made is clear, at least to a human observer with access to the annotation guidelines.

Certain Type B errors are moderately easy to find. When annotators misunderstand a complicated markup guideline, they mismark in a somewhat predictable way. While not being totally systematically detectable, an algorithm can leverage these patterns to extract a list of tags or parses that *might* be incorrect, which a human can then examine. Some errors of this type (henceforth "Type B_1") include:

- VBD / VBN. Often the past tense form of a verb (VBD) and its past participle (VBN) have the same form, and thus annotators sometimes mistake one for the other, as in Figure 12. Some such cases are not detectable, which is why this is not Type A.[6]

- IN / RB / RP. There are specific tests and guidelines for telling these three things apart, but frequently a preposition (IN) is marked when an adverb (RB) or particle (PRT) would be more appropriate. If an IN is occurring somewhere other than under a PP, it is likely to be a mistag.

Occasionally, an extracted list of maybe-errors will be "perfect", containing only instances that are actually corpus errors. This happens when the pattern is a very good heuristic, though not *necessarily* valid (which is why the errors are Type B_1, and not Type A). When filing corrections for these, it is still best to annotate them individually, as the corrections may later be applied to an expanded or modified data set, for which the heuristic would no longer be perfect.

Other fixable errors are pretty much isolated. Within section 24 of the treebank, for instance, we have:

[6]There is a *subclass* of this error which is Type A: when we find a VBD whose grandparent is a VP headed by a form of 'have', we can deterministically retag it as VBN.

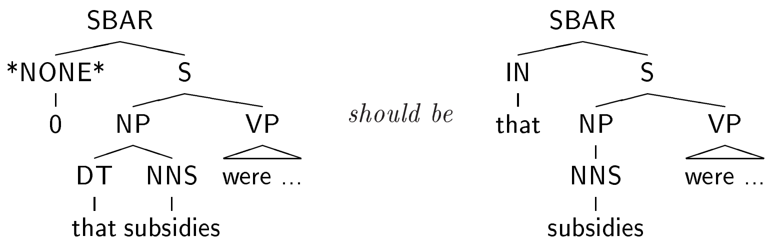

Figure 13: A parse error of Type B_2

- the word 'long' tagged as an adjective (JJ) when clearly used as a verb (VB)
- the word 'that' parsed into a noun phrase instead of heading a subordinate clause, as in Figure 13
- a phrase headed by 'about', as in 'think about', tagged as a location (LOC)

These isolated errors (resulting, presumably, from a typo or a moment of inattention on the part of the annotator) are not in any way predictable, and can be found essentially only by examining the output of one's algorithm, analysing the "errors", and noticing that the treebank was incorrect, rather than (or in addition to) the algorithm. We will call these Type B_2.

8.2.3 Type C: Systematic inconsistency

Sometimes, there is a construction that the markup guidelines writers didn't think about, didn't write up, or weren't clear about. In these cases, annotators are left to rely on their own separate intuitions. This leaves us with markup that is inconsistent and therefore clearly partially in error, but with no obvious correction. There is really very little to be done about these, aside from noting them and perhaps controlling for them in the evaluation.

Some Type C errors in the treebank include:

- 'ago'. English's sole postposition seems to have given annotators some difficulty. Lacking a postposition tag, many tagged such occurrences of 'ago' as a preposition (IN); others used the adverb tag (RB) exclusively.[7] Since some occurrences really are adverbs, this just makes a big mess.
- ADVP-MNR. The MNR tag is meant to be applied to constituents denoting manner or instrument. Some annotators (but not all) seemed to decide that any adverbial phrase (ADVP) headed by an '-ly' word must get a MNR tag, applying it to words like 'suddenly', 'significantly', and 'clearly'.

[7] In particular, the annotators of sections 05, 09, 12, 17, 20, and 24 used IN sometimes, while the others tagged all occurrences of 'ago' as adverbs.

The hallmark of a Type C error is that even what *ought* to be correct isn't always clear, and as a result, any plan to correct a group of Type C errors will have to first include discussion on what the correct markup guideline should be.

8.2.4 `tsed`

In order to effect these changes in some communicable way, we have implemented a program called `tsed`, by analogy with and inspired by the already prevalent `tgrep` search program.[8] It takes a search pattern and a replacement pattern, and after finding the constituent(s) that match the search pattern, modifies them and prints the result. For those already familiar with `tgrep` search syntax, this should be moderately intuitive.

To the basic pattern-matching syntax of `tgrep`, we have added a few extra restriction patterns (for specifying sentence number and head word), as well as a way of marking nodes for later reference in the replacement pattern (by simply wrapping a constituent in square brackets instead of parentheses).

The replacement syntax is somewhat more complicated, because wherever possible we want to be able to construct the new trees by reference to the old tree, in order to preserve modifiers and structure we may not know about when we write the pattern. For full details of the program's abilities, consult the program documentation, but here are the main ones:

- Relabelling. Constituents can be relabelled with no change to any of their modifiers or children.
- Tagging. A tag can be added to or removed from a constituent, without changing any modifiers or children.
- Reference. Constituents in the search pattern can be included by reference in the replacement pattern.
- Construction. New structure can be built by specifying it in the usual S-expression format, e.g. `(NP (NN snork))`. Usually used in combination with Reference patterns.

Along with `tsed` itself, we distribute a Perl program `wsjsed` to process treebank change scripts like the following:

```
{2429#0-b}<<EOF
NP $ [ADJP] > (VP / keep)      (S \0 \1)
NP <<, markets                 - SBJ
EOF
```

This script would make a batch modification to the zeroth sentence of the 29th file in section 24. The batch includes two corrections: the first matches a noun phrase (NP) whose sister is an ADJP and whose parent is a VP headed by the

[8] `tgrep` was written by Richard Pito of the University of Pennsylvania, and comes with the treebank.

word 'keep'. The matched NP node is replaced by a (created) S node whose children will be that very NP and its sister ADJP. The second correction then finds an NP that ends in the word 'markets' and marks it with the SBJ function tag.

Distributing changes in this form is important for two reasons. First of all, by giving changes in their minimal, most general forms, they are small and easy to transmit, and easy to merge. Perhaps more importantly, since corpora are usually copyrighted and can only be used by paying a fee to the controlling body (usually LDC or ELDA), we need a way to distribute only the changes, in a form that is useless without having bought the original corpus. Scripts for `tsed`, or for `wsjsed`, serve this purpose.

These programs are available from our website.[9] More complete documentation on `tsed` can be found in Appendix B.

8.2.5 Training data

In virtually all empirical NLP work, the training set is going to encompass the vast majority of the data. As such, it is usually impractical for a human (or even a whole lab of humans) to sit down and revise the training. Type A errors can be corrected easily enough, as can some Type B_1 errors whose heuristics have a high yield. Purely on grounds of practicality, though, it would be difficult to effect significant correction on a training set of any significant size (such as for the treebank).

Practicality aside, correcting the training set is a bad idea anyway. After expending an enormous effort to perfect one training set, the net result is just one correct training set. While it might make certain things easier and probably will improve the results of most algorithms, those improved results will not be valid for those same algorithms trained on other, non-perfect data; the vast majority of corpora will still be noisy. If a *user* of an algorithm, e.g. an application developer, chooses to perfect a training set to improve the results, that would be helpful, but it is important that researchers report results that are likely to be applicable more generally, to more than one training set. Furthermore, robustness to errors in the training, via smoothing or some other mechanism, will also make an algorithm robust to sparse data (that ever-present spectre that haunts nearly every problem in the field); thus eliminating all errors in the training ought not to have as much of an effect on a strong algorithm.

8.2.6 Testing data

Testing data is another story, however. In terms of practicality, it is more feasible, as the test set is usually at least one or two orders of magnitude smaller than the training. More important, though, is the issue of fairness. We need to continue using noisy training data in order to better model real-world use, but it is unfair and unreasonable to have noise in the gold standard, which causes an algorithm to be penalised where it is more correct than the human annotation.

[9] `http://www.cs.brown.edu/~dpb/tsed/`

As performance on various tasks improves, it becomes ever more important to be able to correct the testing data. A 'mere' 1% improvement on a result of 75% is not impressive, as it represents just a 4% reduction in apparent error, but the same 1% improvement on a result of 95% represents a 20% reduction in apparent error! In the end, a noisy gold standard sets an upper bound of less than 100% on performance, which is if nothing else counterintuitive.

8.2.7 Ethical considerations

Of course, we cannot simply go about changing the corpus willy-nilly. We refer the reader to Chapter 7 of David Magerman's thesis (1994) for a cogent discussion of why changing either the training or the testing data is a bad idea. However, we believe that there are now some changed circumstances that warrant a modification of this ethical dictum.

First, we are not allowed to look at testing data. How to correct it, then? An initial reaction might be to "promise" to forget everything seen while correcting the test corpus; this is not reasonable.

Another solution exists, however, which is nearly as good and doesn't raise any ethical questions. Many research groups already use yet another section, separate from both the training and testing, as a sort of development corpus.[10] When developing an algorithm, we must look at some output for debugging, preliminary evaluation, and parameter estimation; so this development section is used for testing until a piece of work is ready for publication, at which point the "true" test set is used. Since we are all reading this development output already anyway, there is no harm in reading it to perform corrections thereon. In publication, then, one can publish the results of an algorithm on both the unaltered and corrected versions of the development section, in addition to the results on the unaltered test section. We can then presume that a corrected version of the test corpus would result in a perceived error reduction comparable to that on the development corpus.

Another problem mentioned in that chapter is of a researcher quietly correcting a test corpus, and publishing results on the modified data (without even noting that it was modified). The solution to this is simple: any results on modified data will need to acknowledge that the data is modified (to be honest), and those modifications need to be made public (to facilitate comparisons by later researchers). For Type A errors fixed by a simple rule, it may be reasonable to publish them directly in the paper that gives the results.[11] For Type B errors, it would be more reasonable to simply publish them on a website, since there are bound to be a large number of them.[12]

---

[10] In the treebank, this is usually section 24.

[11] The rule we used to fix the LGS problem noted in Section 8.2.1 is as follows:

```
{24*-bg}<<EOF
NP !- LGS > (PP - LGS)   - LGS
PP - LGS                 ! LGS
EOF
```

[12] The 235 corrections made to section 24 can be found in Appendix C.

Finally, we would like to note that one of the reasons Magerman was ready to dismiss error in the testing was that the test data had "a consistency rate much higher than the accuracy rate of state-of-the-art parsers". This is no longer true.

8.2.8 Practical considerations

As multiple researchers each begin to impose their own corrections, there are several new issues that will come up. First of all, even should everyone publish their own corrections, and post comparisons to previous researchers' corrected results, there is some danger that a variety of different correction sets will exist concurrently. To some extent this can be mitigated if each researcher posted both their own corrections by themselves, and a full list of all corrections they used (including their own). Even so, from time to time these varied correction sets will need to be collected and merged for the whole community to use.

More difficult to deal with is the fact that, inevitably, there will be disputes as to what is correct. Sometimes these will be between the treebank version and a proposed correction; there will probably also be cases where multiple competing corrections are suggested. There really is no good systematic policy for dealing with this. Disputes will have to be handled on a case-by-case basis, and researchers should probably note any disputes to their corrections that they know of when publishing results, but beyond that it will have to be up to each researcher's personal sense of ethics.

In all cases, a search-and-replace pattern should be made as general as possible (without being too general, of course), so that it interacts well with other modifications. Various researchers are already working with (deterministically) different versions of corpora—with new tags added, or empty nodes removed, or some tags collapsed, for instance, not to mention other corrections already performed—and it would be a bad idea to distribute corrections that are specific to one version of these. When in doubt, one should favour the original form of the corpus, naturally.

The final issue is not a practical problem, but an observation: once a researcher publishes a correction set, any further corrections by other researchers are likely to decrease the results of the first researcher's algorithm, at least somewhat. This is due to the fact that that researcher is usually not going to notice corpus errors when the algorithm errs in the same way. This unfortunate consequence is inevitable, and hopefully will prove minor.

8.2.9 Experimental results

Next, we compiled all the noted treebank errors and their corrections. The most common correction involved simply adding, removing, or changing a function tag to what the algorithm output (with a net effect of improving our score). However, it should be noted that when classifying reported errors, we examined their contexts, and in so doing discovered other sorts of treebank error. Mistags and misparses did not directly affect us; some function tag corrections actually

Grammatical	Precision	Recall	F-measure
Treebank	96.227%	94.897%	95.558%
Fixed	97.018%	95.137%	96.068%
False error	20.965%	4.703%	11.481%

Form/function	Precision	Recall	F-measure
Treebank	81.224%	76.196%	78.630%
Fixed	86.743%	78.078%	82.182%
False error	29.394%	7.906%	16.621%

Topicalisation	Precision	Recall	F-measure
Treebank	97.115%	93.519%	95.283%
Fixed	99.048%	94.545%	96.744%
False error	67.002%	15.831%	30.973%

Miscellaneous	Precision	Recall	F-measure
Treebank	54.545%	25.000%	34.286%
Fixed	66.667%	30.769%	42.105%
False error	26.668%	7.692%	11.899%

Table 6: Function tagging results, adjusted for treebank error

decreased our score. All corrections were applied anyway, in the hope of cleaner evaluations for future researchers. In total, we made 235 corrections, including about 130 simple retags.

Finally, we re-evaluated the algorithm's output on the corrected development corpus. Table 6 shows the resulting improvements. Precision, recall, and F-measure are calculated as in (Blaheta and Charniak, 2000). The false error rate is simply the percent by which the error is reduced; in terms of the performance on the treebank version (t) and the fixed version (f),

$$\text{False error} = \frac{f - t}{1.0 - t} \times 100\%$$

This is the percentage of the reported errors that are due to treebank error.

The main point to be made here is that the false error rates are much higher for precision than for recall, indicating that the main source of treebank error (at least in the realm of function tagging) is due to human annotators forgetting a tag.

8.2.10 Further notes on error correction

In this section, we have given a new characterisation of the sorts of noise one finds in empirical NLP, and a roadmap for dealing with it in the future. For many of the problems in the field, the state of the art is now sufficiently advanced that evaluation error is becoming a significant factor in reported results; we show that it is correctable within the constraints of practicality and ethics.

Although our examples all came from the Penn treebank, the taxonomy presented is applicable to any corpus annotation project. As long as there are typographical errors, there will be Type B errors; and unclear or counterintuitive guidelines will forever engender Type A and Type C errors. Furthermore, we expect that the experimental improvement shown in Section 8.2.9 will be reflected in projects on other annotated corpora—perhaps to a lesser or greater degree, depending on the difficulty of the annotation task and the prior performance of the computer system.

An effect of the continuing improvement of the state of the art is that researchers will begin (or have begun) concentrating on specific subproblems, and will naturally report results on those subproblems. These subproblems are likely to involve the complicated cases, which are presumably also more subject to annotator error, and are certain to involve smaller test sets, thus increasing the performance effect of each individual misannotation. As the sizes of the subproblems decrease and their complexity increases, the ability to correct the evaluation corpus will become increasingly important.

9 Comparison to prior work

We will now discuss the major differences between our work and the related work reported in Section 2. We omit a comparison with the Collins (1997) paper because it is more distantly related than the others, and does not report any results for the performance of the actual marking of the complement/adjunct distinction.

9.1 Gildea and Jurafsky

Though the high-level task was similar, it is very difficult to compare performance between our work and the work of Gildea and Jurafsky (2000). The actual roles they try to mark, which they call "semantic", correspond roughly to our "grammatical tags" (though not exactly—for example, both a SBJ and an LGS under our system would be marked the same under theirs). Our semantic, or "form/function" tags, primarily mark adjunct phrases, which are not considered by their system. Nevertheless, while our grammatical tags are (relatively) easy to read off the parse trees, their tags are heavily lexicalised within their FrameNet domain/frame/role framework.

Furthermore, where our system looks at every single constituent with a view to possibly function tagging it, their system accepts as input the location of the frame elements to tag. A separate experiment to try and recover these to-be-tagged frame elements got more than a third of them wrong. We hesitate to draw any direct comparisons between results here.

9.2 Brants, Skut, and Krenn

The work of Brants et al. (1997) is much easier to compare, however. While there is not an exact match between the information recovered by the two systems, there is a great deal of overlap; with certain caveats, we are prepared to compare these results with our grammatical tagging results.

In their system, all constituents are function-tagged. This, of course, eliminates any precision-recall tradeoff issues they might have had. It makes their job a bit harder, in that they have more tags (17) to work with; but it also makes their job a bit easier, because many tags will be easy to predict from context. The Penn treebank, by and large, prefers to leave such constituents untagged.

For instance, we note that their best performance is on tagging the children of PP nodes (on which their accuracy was 97.9%, and which comprise 24% of the data). This is almost certainly because the possible expansions of a PP are relatively constrained—it appears that the vast majority of prepositional phrases in their system will expand to an AC ('adpositional case marker') followed by one or more NKs ('noun kernels'). Comparable phrases in the Penn treebank will have no function tags at all.

A better comparison might be drawn between our F-score (95.9%) and their performance on children of S nodes (89.1%, comprising 26% of the data). Here there are myriad expansion possibilities, and rather more possible function tags to apply. Three of the four tags shared between the two systems (our SBJ, DTV, PRD; their SB, DA, PD) will in their system almost always be found on children of S nodes.

Perhaps, in the end, the best comparison would be between their overall accuracy (94.2%) and our 'with-null accuracy' percentage (99.0%), since it takes into account every time we correctly decide to not tag something, which corresponds to a decision in their system to tag a constituent with a non-Penn-treebank tag. On the other hand, this effectively collapses all non-Penn-treebank tags into one tag for us but not for them, which would give us an unfair advantage.

A summary of these results can be seen in Figure 14.

10 System improvements

We have presented a method for assigning function tags to parsed text. While it is a great improvement on previously-extant work, a number of ideas present themselves as possible improvements worth trying. One major class of improvements is methodological, both in incremental changes to the basic feature tree system and in trying entirely different systems.

10.1 Different feature trees

Although we tested more than a thousand feature trees besides those given in Figures 4 through 8, the exploration was only partially automated. A more

Blaheta	
No-null precision	96.5%
No-null recall	95.3%
No-null F-measure	95.9%
With-null accuracy	99.0%

Brants, Skut, and Krenn	
PP children	97.9%
S children	89.1%
Overall accuracy	94.2%

Figure 14: A comparison between our work and Brants et al

To parse a constituent, given its label and the labels of all its siblings, ancestors, and siblings of ancestors, and the heads of all its ancestors,

- Guess the POS of the head (if not already known)
- Guess the head (if not already known)
- Guess the label of the head-containing child (HCC)
- Guess the labels of all children to the left of the HCC
- Guess the labels of all children to the right of the HCC
- Recursively parse all the children

Figure 15: The head-driven algorithm used in the parser

systematic investigation into the advantages of different feature trees would be useful; a system that automatically learned the best feature trees would be ideal. Due to computational limitations, even automated this task would take a very long time, especially as we begin to try adding a large number of new features into the mix. As such, we hope to defer it until after some other systems have been tried.

10.2 Integration into parsing

There is no reason to think that this work could not be integrated directly into the parsing process. Currently, we wait for the output of the parser, take it as correct, and then simply assign the most likely function tag to each individual constituent. However, since the current parser is generative (see Figure 15), it should be relatively straightforward to add the function tagging right in: we

	Precision	Recall
Grammatical		
Feature trees	98.21%	97.36%
Decision trees	98.75%	97.55%
Form/function		
Feature trees	80.61%	76.88%
Decision trees	81.08%	71.07%

Figure 16: Preliminary decision tree results

could simply introduce the function tags as another feature to guess, whose probability would be multiplied into the overall distribution. This information could then prove quite useful within the parse itself, to rank several parses to find the most plausible.

10.3 Decision trees

A decision tree system is, like our system, based on asking questions about certain features of a given constituent. However, instead of asking the same questions all at once, a decision tree chooses which questions to ask *based on* the answers to the previous questions. The tree structure here arises differently from that of feature trees: before any questions are asked, there is a single root state, and as questions are asked and answered the process descends through the branches of the tree. At the leaf nodes we find the decision that the tree has made, the value it chooses for the feature in question.

It seems like this ability to change the conditioning features based on the values of other conditioning features would come in very handy. For instance, on the one hand a constituent is very likely to be a predicate (PRD) if it is a noun phrase under a verb phrase headed by a form of '`to be`', irrespective of what the head or siblings of the noun phrase might be; on the other hand, a noun phrase that is the left sister of a verb phrase is likely to be a subject (SBJ), whatever the verb in the sentence might be.

The difficulty, of course, is in automatically generating these decision trees. (Generating them by hand would be even harder than making feature trees by hand, since the number of nodes is much higher. It is thus simply not feasible, so we move straight to automation.) There are a number of algorithms to do this that we would like to explore. Ross Quinlan has implemented one of these in a program called `c4.5`; we would like to use and adapt this system for function tagging. Preliminary testing of `c4.5` has been promising—we trained on 1 million constituents of the training data, then tested on the (uncorrected) development corpus. Results are reported in Figure 16. (Note that the feature tree results, provided for comparison, are on the treebank parses of the development corpus, since that is also what the decision tree tagging was based on.) Performance on the grammatical tags is improved very slightly over the feature tree system. The form/function tags fare less well; the precision goes up slightly, but the recall loses more than five points. We suspect that this is largely a result of the smaller training corpus, and hope to remedy that.

10.3.1 Performance

We would like to modify `c4.5` (or some other system) to handle larger datasets and larger feature sets. The 1M-constituent training limit is due to memory limitations, but since the total training is about 1.7M, full training seems within reach.

10.3.2 Other input

In order to be directly comparable to the feature-tree system, the decision-tree version should be run on parsed output (the preliminary results were run on the hand-parsed treebank), for the reasons given in Section 5. We should also definitely re-run the system on the corrected development corpus—a 1% improvement on the grammatical tags would correspond to an error reduction of *half*.

10.4 Neural nets

A neural net is a relatively simple probabilistic model that, again, makes use of features to decide a value for some variable. The perceptron algorithm goes as follows: all of the (boolean) features of a given constituent—including the feature being predicted—are placed into a vector. Of course, since we don't yet know the final predicted value, we have a number of candidate vectors at this point. These vectors are multiplied by a vector of weights and summed to find a score for each candidate; the candidate with the highest score is then predicted. In the training stage, this predicted value is compared against the actual value, and if the prediction was incorrect, then the vector for the correct candidate is added right into the weight vector while subtracting out the incorrectly predicted candidate.

There has been a recent resurgence of interest in neural nets; Michael Collins Collins and Duffy (2002; Collins (2002) has shown that the classic perceptron model (Rosenblatt, 1958) and the newer voted perceptron model (Freund and Schapire, 1999) can be used to great effect in the domains of part-of-speech tagging, parsing, and named-entity extraction. It certainly seems as if this is worth trying in the function tagging domain as well.

A big advantage of neural nets is that they deal well with having large numbers of features. Since a major focus of our work will be trying other features and deciding which ones are useful (see Section 11 below), this should prove to be very important.

10.5 Subcases

We don't need to just automatically recommend the highest-scoring function tag. Since we have a probability score associated with this high-scoring tag, we can evaluate what to do based on this score. In particular, if it is sufficiently close to the next-most-likely tag, we can ask the human being who is running the program which one to choose. (Something similar is done in (Brants et al., 1997), but the mechanism needs to be different here.) Ideally, the training could be updated in some way with this new tagging.

More generally, we could provide a formal mechanism for adding data on specific cases: i.e. lots of conjunction examples, conditioned on the presence of 'and'. This is very tricky, however, as simply adding the examples to the training would skew the statistics on unrelated cases.

11 Additional features

Another major area of improvement is in the features that are used. A casual analysis of the errors produced by our system yielded a number of examples where the system simply didn't have access to the information it would have needed to correctly distinguish two cases. Thus, we have a number of ideas for additions and changes to the features used by the system.

11.1 Object heads

When we first naïvely defined the notion of an "alternate head", we said that it was the second child of a PP. However, if there is a premodifier (such as 'only' or 'an hour'), the second child is the preposition itself, which is not really what we want. We want the head of the object of the preposition. We will add a feature for these true "object heads".

Furthermore, it may be of interest to try to calculate object heads for more than just PPs. A first pass at defining a rule for determining object heads: first calculate the head, and note which child of the current node contains it. Then the object head will be the head of the subsequent child, if any, or else the preceding child. For syntactic categories that are often headed by closed-class words, like SBAR or even VP (which are often headed by modals or auxiliary verbs), the object head may be able to give a better handle on the semantic import of the phrase.

11.2 Conjunctions

Right now, anything with two conjoined phrases is often handled wrong: function tagging should occur on the top of the conjunction, but often occurs on the individual members. Our system already assigns different labels to some conjunction nodes: an NP that results from the conjunction of two or more NPs, for instance, is internally relabelled CCNP. Our problem with mistags on conjoined PPs might be solved by an internal CCPP label. Alternatively, it may be helpful to instead create a new CC feature. (The function tag of the parent (see 11.4 below) might be helpful here as well.)

11.3 Multiple siblings

In a little over half the cases where two NPs separated by a comma are directly under an S node,[13] the tagger tags *both* of them as subjects. This is because the two noun phrases have nearly identical contexts; if, however, the first were able to "see" more than the sibling to its immediate right (which is just a comma), it would better be able to judge the situation.

Thus in this and other cases, it might be helpful to have a feature for the label of the second or third node to either side of a given node.

11.4 Other function tags

We can only assign one function tag per group, but sometimes tags in different groups are mutually exclusive as well. For instance, a 'by' phrase can indicate either an LGS or a TMP, but not both. Furthermore, a tag on one node might be dependent on a certain tag on another node, as with the rule that TMP only appears on a PP object if the PP itself is marked DIR. It should therefore help to see the other tags applied to a given constituent and its relatives.

11.5 Lexical backoff

Sparse data is ever a problem, most especially with the lexical items. Where part of speech is often too coarse (e.g. at distinguishing 'at noon' (TMP) from 'at school' (LOC)), the exact lexical item might not be present. As such, it would be good to have many different backoffs for the lexical items.

We could use a variety of different types of backoff. The word's stem (calculated using the Porter algorithm, or a database such as CELEX) would collapse morphological and syntactic distinctions between different forms of a word. To get at semantic groupings, we could use a word's wordnet parent (or some other ancestor), some other hand-built semantic clustering system, or possibly an automatically generated set of semantic clusters. One form of syntactic backoff is the part-of-speech tag, which we already use, but we could in theory back it off further; for instance, we could collapse NN, NNS, NNP, and NNPS into the same 'supertag'.

11.6 Window contexts

It seems as if the sense of a word or phrase can sometimes be guessed just by the mere presence of certain other words nearby. It would therefore be useful to have a feature that includes an entire window of lexical items, to capture lexical dependencies that are in hard-to-define (or rare) syntactic relationships. Formally, this is very straightforward: since all features are really binary, the

[13] tgrep search pattern: `S < (NP $. (, $. NP))`

new window features consist simply of the binary question "does word X appear within ten words of this constituent?"

This presents a technical difficulty, however; in the code and in the discussion, we have relied on the fact that only one of a large set of binary features can be true, so that our representation is just an index recording which single bit is on. But for a window context feature, multiple bits could be on. Systems that expect a relatively small number of features (some of which have 40,000 potential values) may not deal well with an orders-of-magnitude increase in number of features. Implementing this in a way that is computationally feasible is an interesting problem that we look forward to solving.

12 Conclusion

Function tags have in the past not been very well studied or exploited. I have created a system to recover these tags, using a "maximum-entropy inspired" feature tree framework, and found a set of feature trees to use with it that performs with a reasonably high level of accuracy. In analysing the data, I discovered a number of errors in the Penn treebank, and suggested a taxonomy for classifying them and a methodology for fixing them.

I propose to further develop this system by trying a few different algorithmic frameworks—in particular, one based on decision trees, and one based on neural nets. More importantly, using these algorithmic frameworks I propose to test a number of different new types of features. I am confident that at least some of them will be helpful, and it should prove interesting to study and find the linguistic situations in which they help the most.

References

Adam L. Berger, Stephen A. Della Pietra, and Vincent J. Della Pietra. 1996. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71.

Ann Bies, Mark Ferguson, Karen Katz, and Robert MacIntyre, 1995. *Bracketing Guidelines for Treebank II Style Penn Treebank Project*, January.

Don Blaheta and Eugene Charniak. 2000. Assigning function tags to parsed text. In *Proceedings of the 1st Annual Meeting of the North American Chapter of the Association for Computational Linguistics*, pages 234–240.

Don Blaheta. 2002. Handling noisy training and testing data. In *Proceedings of the 7th conference on Empirical Methods in Natural Language Processing.*

Rens Bod. 2001. What is the minimal set of fragments that achieves maximal parse accuracy? In *Proceedings of the 39th annual meeting of the Association for Computational Linguistics*, pages 66–73.

Thorsten Brants, Wojciech Skut, and Brigitte Krenn. 1997. Tagging grammatical functions. In *Proceedings of the 2nd conference on Empirical Methods in Natural Language Processing*.

Eugene Charniak. 1997. Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 598–603, Menlo Park. AAAI Press/MIT Press.

Eugene Charniak. 1999. A maximum-entropy-inspired parser. Technical Report CS-99-12, Brown University, August.

Eugene Charniak. 2000. A maximum-entropy-inspired parser. In *Proceedings of the 1st Annual Meeting of the North American Chapter of the Association for Computational Linguistics*.

Eugene Charniak. 2001. Immediate-head parsing for language models. In *Proceedings of the 39th annual meeting of the Association for Computational Linguistics*, pages 116–123.

Ciprian Chelba. 2000. *Exploiting syntactic structure for natural language modelling*. Ph.D. thesis, Johns Hopkins University.

Mahesh V. Chitrao and Ralph Grishman. 1990. Statistical parsing of messages. In *DARPA Speech and Language Workshop*, pages 263–266.

Michael Collins and Nigel Duffy. 2002. New ranking algorithms for parsing and tagging: kernels over discrete structures, and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 16–23.

Michael Collins. 1999. *Head-driven statistical models for natural language parsing*. Ph.D. thesis, University of Pennsylvania.

Michael Collins. 2002. Ranking algorithms for named-entity extraction: boosting and the voted perceptron. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*.

Yoav Freund and Robert E. Schapire. 1999. Large margin classification using the perceptron algorithm. *Machine learning*, 37(3):277–296.

Daniel Gildea and Daniel Jurafsky. 2000. Automatic labeling of semantic roles. In *Proceedings of the 38th annual meeting of the Association for Computational Linguistics*.

Julia Hockenmaier and Mark Steedman. 2002. Generative models for statistical parsing with combinatory categorial grammar. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*.

David M. Magerman. 1994. *Natural language parsing as statistical pattern recognition*. Ph.D. thesis, Stanford University, February.

David M. Magerman. 1995. Statistical decision-tree models for parsing. In *Proceedings of the 33rd annual meeting of the Association for Computational Linguistics*.

Adwait Ratnaparkhi. 1999. Learning to parse natural language with maximum-entropy models. *Machine learning*, 34:151–178.

Brian Roark. 2001. Probabilistic top-down parsing and language modelling. *Computational linguistics*, 27(2):249–276, June.

Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65:386–408. Reprinted in *Neurocomputing* (MIT Press, 1998).

A Function tags

For a more complete description of each function tag, with more comprehensive examples, see Section 2.2 of (Bies et al., 1995).

A.1 Grammatical tags

These six tags serve primarily to mark the grammatical role a given constituent plays in a sentence. Verb phrases are assumed to be predicates, and are therefore unmarked. Noun phrases at the sentence or verb phrase level that have neither grammatical nor form/function tags are objects (either direct or indirect, but see A.1.1).

These tags correspond to those presented in Section 2.2.2 'Grammatical role' in (Bies et al., 1995), except that that section also includes topicalisation, which we have separated into a separated category (see A.3).

A.1.1 DTV—Dative

This tag marks indirect objects in their prepositional form. For this tag to occur, the verb needs to allow the indirect object to appear either before the direct object ('shifted') or in a 'to'-phrase after the direct object ('unshifted'); and the unshifted form must be the one used. (Shifted datives are unmarked.) In the sentence "Alex gave Sasha a book", 'Sasha' is the indirect object but does *not* get a DTV tag; however, in the shifted version "Alex gave a book to Sasha", the phrase "to Sasha" is marked DTV.

If the preposition used is 'for', and the object appears in either shifted or unshifted form, the BNF tag is used (see A.2.2).

A.1.2 LGS—Logical subject

If a sentence is passive, then the subject of the sentence is what would be the object in an active version of the sentence. The subject of the active version, if it's present at all, is tucked inside a 'by'-phrase under the verb. This "logical" subject is then marked LGS. In the sentence "Alex was hit by Sasha", 'Sasha' would be marked LGS.

It is important to note that the tagging guidelines specifically state that the LGS tag attaches to the NP object of 'by', and not to the PP itself; it is also important to note that the human annotators screwed this up with some regularity (see Section 8.2.1).

A.1.3 PRD—Predicative

Any predicative construction that is not a VP is marked with this tag. For instance (and most commonly), this includes noun and adjective phrase objects of 'be'. In most cases where the PRD-marked phrase is not itself an object of a stative verb like 'be', it heads a VP-less S node, as in "Alex kept the party a secret", where "the party" is the subject and "a secret" the predicate of the S node object of the VP headed by 'kept'.

A.1.4 PUT—Locative complement of 'put'

This (relatively rare) tag marks the locative complement of 'put'. Since 'put' in its most basic meaning requires more than just a direct object (cf "*Sasha put the book."), but the required location is not a direct/indirect object, it needs to be marked. Thus in "Sasha put the book on the web", "on the web" is marked PUT.

Note, however, that other verbs that behave similarly (e.g. 'set': "*Sasha set the book") do not get this tag.

A.1.5 SBJ—Subject

The subject of every S node gets this tag. As nearly every "sentence" in the treebank contains one S (or SINV, etc) at the top and often a few nested, this is by far the most common function tag in the treebank.

A.1.6 VOC—Vocative

This tag is marked on phrases of address; in "Sasha, read the book", 'Sasha' would be marked VOC. Since news articles rarely address the reader directly, the Penn treebank contains very few instances of this tag (mostly in quotations).

A.2 Form/function tags

This category includes those tags introduced in Sections 2.2.1 and 2.2.3 of the bracketing guidelines. They tend to mark the semantic role of a constituent, but there are few universal statements that can be made about their usage. In many cases they mark a discrepancy between the form and the function of a constituent (thus the name), though not always. In many cases they act adverbially, but even aside from the NOM tag they sometimes modify noun phrases or other things. Most mark adjuncts, but some mark complements.

The name "form/function" is taken from the title of Section 2.2.1 of the bracketing guidelines. It's not great for this category, but we haven't heard any better ones, so it remains.

A.2.1 ADV—Adverbial

Constituents are marked ADV if they are not adverbial phrases, but are acting adverbially, but aren't marked with some other form/function tag. For instance, in "Sasha felt slightly sick", "slightly" is an adverb comprising an adverbial phrase, and therefore would not be tagged ADV. But in "Sasha felt a little bit sick", "a little bit" is a noun phrase acting adverbially, so it is tagged ADV. On the third hand, in "Sasha grew a little bit", "a little bit" is still modifying the verb, but it has a more specific tag to use (EXT), so it doesn't get an ADV tag.

A.2.2 BNF—Benefactive

This tag marks an indirect object, in either shifted or unshifted form (see A.1.1), that uses or would use the preposition 'for'. That is, in the sentences "Alex made Sasha a pie" and "Alex made a pie for Sasha", "Sasha" and "for Sasha" (respectively) would carry the BNF tag.

A.2.3 DIR—Direction

Constituents—usually prepositional phrases—that answer the questions "from where?" and "to where?" are marked with the DIR tag. (Those that answer the question "through where?" are marked LOC instead.)

As with many other tags, this includes metaphorical location, especially financial, as in "The price rose to $37", wherein "to $37" would be marked DIR.

A.2.4 EXT—Extent

This marks the (usually financial 'pseudo-spatial') extent of an activity, as in "The price rose $4", where "$4" would be marked EXT.

Unlike most of the other tags in this category, EXT is rarely attached to a prepositional phrase, more usually marking noun phrases.

A.2.5 LOC—Locative

This very common tag is used to mark phrases that denote the place where something takes place, usually as a VP modifier but also sometimes found on an NP modifier.

The location may be metaphorical, as in "panic in the market" or "banking at that company". However, idiomatic location, as in "under pressure", is not marked LOC.

If the location is a source or a destination, DIR may be more appropriate.

A.2.6 MNR—Manner

This tag is used for phrases that indicate the manner in which some action was performed, or the instrument with which it was performed.

Some annotators liberally mark nearly every otherwise-unmarked ADVP as MNR, while others only mark phrases that actually indicate manner (see Section 8.2.3).

A.2.7 NOM—Nominative

When headless relative clauses and gerunds are used nominally, they are marked NOM. (For instance, in "Baking pies is fun", "baking pies" is NOM.) Other non-NPs are not so tagged.

A.2.8 PRP—Purpose

This tag marks constituents that annotate the purpose of or reason for an action. In the sentence "Sasha ran for health reasons", "for health reasons" would be marked PRP.

A.2.9 TMP—Temporal

This is the most common tag in this category, and the second most common overall; it marks temporal constituents, those which answer the questions "when?", "how often?", and "how long?". Temporal phrases can be either noun phrases or prepositional phrases, but in a choice between marking a PP or its NP object, annotators are advised to mark the PP.[14]

A.3 Topicalisation (TPC)

This tag marks elements that precede the subject in a declarative sentence. Topicalised elements must meet other criteria, which are given in the bracketing guidelines.

The reason this tag was separated from the other grammatical tags is that it can co-occur with several of them. Since the categories are comprised of otherwise mutually exclusive tags, it seemed a good idea to remove this one.

A.4 Miscellaneous

These tags don't fit into the other categories.

A.4.1 CLF—'It'-Cleft

Marks cleft constructions. In the sentence "It was Sasha that read the book", the top-level S node would be marked CLF.

A.4.2 CLR—"Closely related"

The CLR tag was meant to mark a variety of phenomena, most of which could be loosely grouped as collocation and idiom. The largest subgroup of CLR markings are phrasal verbs like 'rely on'.

The problem with the tag is that it is very vaguely defined, and its annotation varies considerably from person to person. As a result, we have largely disregarded it here.

A.4.3 HLN—Headline

Headlines (and datelines) are marked with HLN; such constituents always occur at the top level, distinct from the following sentence.

[14] Except if the PP is already marked DIR and the sense of the expression is financial.

A.4.4 TTL—Title

This tag marks the titles of books, paintings, shows, and so on. Titles may be of any constituent type, and may be altogether ungrammatical—the rare labels NAC, NX, and X (all of which indicate unusual and difficult-to-bracket grammatical constructions) each occur at least once in the Penn treebank with TTL.

B tsed

B.1 The command-line

`tgrep [options] `*`spattern file(s)...`*

Prints to standard output those sentences in *`file(s)`* that match *`spattern`*. *`spattern`* is a search pattern constructed as described in Section B.2. Note that for most nontrivial search patterns you will need to put single quotes around the search pattern so that your shell does not interpret it as multiple arguments. Note also that in some shells (e.g. `tcsh`), some characters (e.g. `!`) will be grabbed by the shell even when single-quoted; this can be worked around with the alternate operators discussed in Section B.2.3.

`tsed [options] `*`spattern rpattern file(s)...`*

Prints to standard output the entirety of *`file(s)`*, with every sentence that matches *`spattern`* modified according to *`rpattern`*. *`spattern`* is a search pattern as for `tgrep`, described in Section B.2. *`rpattern`* is a replacement pattern, detailed in Section B.3.

`tsed [options] -b[`*`patfile`*`] `*`file(s)...`*

Instead of providing a single search-and-replace pattern on the command line, you can provide any number of them in a file (*`patfile`*, if present) or on standard input. Each line of the batch file contains one spattern (Section B.2) and one rpattern (Section B.3), each enclosed in curly brackets.[15] `tsed` will iterate through the patterns on a per-sentence basis, reading the sentence only once and printing it out when done.

`wsjsed [options] `*`file(s)...`*

When making a moderate number of changes to a corpus, or when making changes you'll want to communicate to someone else, you should wrap `tsed` with some program that understands the file structure of that corpus. For the Penn treebank, we have written `wsjsed`; it is described in Section B.4. Scripts to `wsjsed` can be provided as filenames on the command line or directly to standard input.

[15] The format is slightly more forgiving than that: any amount of whitespace can precede, follow, or separate the bracketed patterns, and the whole thing may be followed by a '`#`'-delimited comment. It must all fit on one line, however.

B.1.1 General options

Note that (for now) all options to `tgrep` and `tsed` need to be passed separately—"`tsed -C -g ...`" rather than "`tsed -Cg ...`"—for (not very good) historical reasons. This will be fixed in a future release.

`-D`*dir* The directory where internal data files reside. These include e.g. `headInfo.txt`, which is used to implement the head-finding `/` operator.

`-I`*dir* If present, this argument will be prepended to each input file provided.

`-d`*N* If *N* is greater than zero, provides arcane debugging output.

`-C` Causes matching of leaf nodes (i.e. words) to be case-insensitive.

`-h` Provides a list of command-line options.

B.1.2 `tgrep` options

`-f` Prints file and sentence number information for every match.

`-s` Instead of printing each match by itself, prints the entire sentence containing (at least) one match.

B.1.3 `tsed` options

`-g` Replace all occurrences of a pattern. In batch mode, this option applies on a per-pattern basis; that is, the first spattern is matched and all occurrences are modified according to the rpattern before considering the second pattern.

`-G` Perform replacement, then repeat search-and-replace. This differs from the `-g` option in that for `-g` all the matches are found before any of the replacements are performed, and it is therefore guaranteed to terminate. The `-G` option, on the other hand, will re-match the spattern after the rpattern is applied, until no change results. Especially when used together with the `-b` option, it is very possible to cause infinite loops! The `-G` and `-g` can be used together, although in *most* (not all) cases the `-g` will be redundant.[16]

`-i` Perform change in place rather than outputting to standard output. Most corpus modification (e.g. `wsjsed`) will use this option.

`-c` Only print those sentences where at least one change was made. If a pattern fires but the modification is null (e.g. removing a tag that isn't there), the sentence won't be printed.

[16] The application of the rpattern on an early match could cause a later spattern match to fail; if the `-g` option is used, all the spattern matches will be found already, and the rpattern applications won't affect them. If both `-g` and `-G` are used, all the spattern matches will be found, and the rpatterns will be applied; and then the whole thing will be repeated to see if any new spattern matches were formed.

B.1.4 `wsjsed` options

`-w`*path* Tells `wsjsed` where to find the copy of the Penn treebank to be modified. Default is the relative path ‘`wsj/`’.

B.2 Search patterns

The search pattern syntax is based heavily on that used in Richard Pina’s `tgrep` that comes packaged with the Penn treebank. There are some major differences, however, which will be noted below.

The basic structure of a search pattern is as follows:

base [*op arg*]*

The *base* of the pattern is a label or word to be searched for; a pattern with nothing but a base label will print every single tree that is rooted at a node with that label. To have a base match anything, use the underscore (‘_’) wildcard.

After the base you can have any number of *restrictions*, each of which are composed of an *operator* and an *argument.* The operators must be separated from the preceding pattern and from the succeeding argument by whitespace; this is because virtually any punctuation can actually occur in a word, and thus whitespace is our only consistent delimiter. (Also parentheses. More on that in a minute.) Some operators take atomic things, like numbers and labels, as their arguments; others can take patterns. Unless they are comprised solely of a base pattern, however, they must be enclosed by parentheses or square brackets, as the next unparenthesised operator will be taken as beginning a new restriction. For instance,

```
VP < NP < PP
```

searches for a VP that has at least one NP child and at least one PP child; on the other hand

```
VP < (NP < PP)
```

searches for a VP that has at least one NP child *which itself* has at least one PP child. Patterns can be nested indefinitely, and some pretty complex ones can arise; examples of this can be seen in Section B.2.4. The difference between parentheses and square brackets is irrelevant for `tgrep`; the use of square brackets in `tsed` spatterns is discussed in Section B.3.1.

It should be noted that search patterns are considered from left to right; for every constituent, it is first compared against the base pattern, then against each restriction in turn. If at any point a comparison fails, the remainder are not considered. It is therefore prudent to put any compute-intensive restrictions at the end of the pattern.

B.2.1 Search pattern operators

Here we detail the behaviour of each operator, and say what kind of argument it can take.

Sentence number: #

This operator accepts a numeric argument n, and automatically fails unless the constituent under consideration is in sentence n of the current input file (numbered from 0). Thus

```
S # 4
```

would find all S nodes in sentence 4 (the 5th sentence) of any of the input files. This will normally only be run on individual files, and is probably only useful in error-correction `tsed` patterns.

Function tag: -

This operator accepts a tag x, and succeeds if x is among the hyphen-separated tags modifying the constituent under consideration.

```
PP - TMP
```

finds all PP nodes tagged with the TMP function tag.

Head: /

This operator accepts a word w, and succeeds if w is the head word of the constituent under consideration. NB: the algorithm used to determine head words is specific to the Penn treebank tagset, and will work unpredictably or not at all with other tagsets!

```
NP / stock
```

finds the noun phrase(s) headed by the word stock.

Note that all heads are calculated when trees are read in, so that this restriction can be checked in constant time. Unfortunately, this imposes a penalty on spatterns that don't require it, so it may change in future versions (so that heads are only calculated if they will be used, for instance).

Ancestor: >>

This operator accepts a pattern p, and succeeds if any node on the path from the current constituent to the root node, including the root node but not the current constituent, is matched by the pattern p.

```
PP >> S
```

finds any prepositional phrase (PP) anywhere below an S node.

Parent: >

This operator accepts a pattern p, and succeeds if the parent of the node under current consideration is matched by p.

```
_ > PRN
```

finds all children of parentheticals (PRN).

Child: <

This operator accepts a pattern p, and succeeds if *any* child of the node under current consideration is matched by p.

```
PP < DT
```

finds all prepositional phrases that contain determiners at their top level.

Specific child: `<N`

This operator accepts a pattern p, and succeeds if the Nth child of the node under current consideration is matched by p. An N of 1 means the leftmost child. If N is negative, count proceeds from the right, so that an N of -1 means the rightmost child. The operators `<,` and `<‘` are synonyms for `<1` and `<-1`, respectively.

```
S <‘ NP
```

finds an S-node whose *last* child is a noun phrase.

Descendant: `<<`

This operator accepts a pattern p, and succeds if a node matching p can be found anywhere in the subtree rooted at the node under current consideration.

```
S << PP
```

finds an S-node that has *any* PP descendants.

Left descendant: `<<,`

This operator accepts a pattern p, and succeeds if any left descendant of the node under current consideration matches p. A left descendant is any descendant that can be reached only by going through the leftmost children of each node in turn.

```
S <<, Each
```

finds any S-node whose first word is 'Each'.

Right descendant: `>>‘`

This operator accepts a pattern p, and succeeds if any right descendant of the node under current consideration matches p. A right descendant is any descendant that can be reached only by going through the rightmost children of each node in turn.

```
S <<‘ ?
```

finds any S-node that ends with a question mark.

Sibling: `$`

This operator accepts a pattern p, and succeeds if any sibling of the node under current consideration matches p. Note that the siblings can occur in either order.

```
NP $ PRN
```

finds a noun phrase that is sibling to a parenthetical.

Right-adjacent sibling: `$.`

This operator accepts a pattern p, and succeeds if the sibling following the node under current consideration matches p. This operator is sensitive to order.

```
NP $. ,
```

finds a noun phrase that is followed by (not preceded by!) a comma-like punctuation mark. (That is, anything whose part-of-speech tag is a comma.)

Precedent sibling: `$..`

This operator accepts a pattern p, and succeeds if any sibling that precedes the node under current consideration matches p. The precedent sibling need not be adjacent.

```
PP $.. VP
```

finds a prepositional phrase preceded by a sibling VP.

B.2.2 Search pattern negation: `!`

Any operator can be negated by preceding it with an exclamation point. Under the hood, this is implemented as simply running the restriction as if it were not negated, then reversing the sense of the result.

```
IN !> PP
```

finds prepositions not directly under a prepositional phrase.

B.2.3 Alternate operator characters

Many patterns will be entered on the command line, but shells often do not play well with operator characters (notably '!'). As a result, there are some substitutions that can be made:

N	!
S	$
P	<
C	>

Table 7: Operation substitution characters

These can be made on a character-by-character basis; for instance, legitimate substitutes for `!<<` include `N<<`, `NPP`, and even `NP<`.

B.2.4 Search pattern examples

Here we list a few different sample search patterns, with a prose description of each.

Mislabelled logical subjects

```
NP > (PP - LGS)
```

This pattern finds all noun phrases whose parents are prepositional phrases marked as logical subjects. Note that since the base pattern is the NP, it is that node that will be printed out (not the parent PP). To print the PP instead, we would write the spattern as

```
PP - LGS < NP
```

or

```
PP < NP - LGS
```

which are semantically equivalent. The first will be somewhat faster, however, as restrictions are processed in order: the tag restriction string-compares "LGS" against every tag on that PP. If there are no tags, it fails immediately; in any case it's unlikely to have more than one tag to compare against. On the other hand, the child restriction needs to iterate through all children (always at least one, generally two or more) and string-compare "NP" against all the labels. This cost is more pronounced for constituent types that tend to have many children, and it's much much more pronounced for the descendant restriction (<<), which has to check the entire subtree.

'from...ago' phrases

```
PP / from <<` ago
```

This pattern finds all prepositional phrases headed by 'from' and ending with 'ago'. The head test is cheap (see Section B.2.1), and the right-descendant test is not terribly expensive—linear in the depth of the subtree, rather than in the total size of the subtree. This won't get as many results as

```
PP / from << ago
```

but if you only care about phrases that *end* with 'ago', the former will be substantially more efficient.

B.3 Replacement patterns

Once the pattern is matched, we need to know what to replace it with. The different types of replacement patterns tend to change a minimum of things about the matched pattern—no single rpattern can change both the label and the function tag of a node, for instance. As a result, a number of fairly simple changes will require multiple spattern-rpattern pairs to be applied in sequence. At some point there may be a mechanism for applying multiple rpatterns to a single spattern, but as there is already a batch capability in `tsed`, this is a relatively low priority.

B.3.1 Referring to the matched spattern

There are several places in an rpattern where you might need to refer to something in the corresponding spattern. This is done by using square brackets [] instead of parentheses () in the spattern. The subpattern is referred to by a backslash followed by a number; the number corresponds to the sequential number of the square-bracketed subpattern. Such subpatterns are counted from 1, and in order by their left (open) bracket. Thus in a pattern like

```
A < [B < [C]] < [D]
```

we have `\1` referring to the B node, `\2` to the C node, and `\3` to the D node. The zeroth match, `\0`, refers to the entire spattern match (here the A node).

B.3.2 The rpattern types

Removing nodes: `()`

To remove a matched node from the tree it is in, the rpattern is simply an empty pair of parentheses.

Relabelling nodes: *`LABEL`*

To change the label of the matched node, the rpattern is just the label to replace it with. Using this type of rpattern will not change the children of a given node, or any of the tags associated with it.

Retagging nodes: `-` *`TAG`*`,` `!` *`TAG`*

To add a tag to a node, use a hyphen, a space (that's important), and the tag to be added. To remove a tag, replace the hyphen with an exclamation point. Because it's possible to have multiple tags on a node, actually *changing* the tag is done by removing the old one and adding the new one—merely adding the new tag will preserve the old one intact. If a tag is to be added but is already present, no error is generated, but the program considers that no change is made (which may generate a warning down the line). Likewise, if a tag is to be removed but is not present, `tsed` considers that no change is made.

Renumbering nodes: `-` *`n`*`,` `!` *`n`*

To add or change the coreference index of a node, use a hyphen, a space, and the new coreference index. To remove the index, replace the hyphen with an exclamation point. It is not possible to have multiple coref indices for a single node, so adding a number to an already-indexed node will change the number. If an index is to be added and that n is already the index of the node, the program considers that no change is made. If an index is to be removed and no index is present, or the index is different from n, no change is made.

New subtrees: `(`*`LABEL`*` ...)`

If the matched node is to be replaced with all-new subtrees that were not present in the original tree, the usual treebank S-expression format is used: an open paren, the label of the node, all the children of the new node, and a close paren. The children can themselves be "new" subtrees like this one, or "reference" or "reuse" subtrees (see below).

Referencing subtrees: `\`*`n`*

To include a submatch in its entirety with no alteration, just refer to it as described in Section B.3.1.

Reusing subtrees: `(\`*`n`*` ...)`

If there is a submatch of the spattern that contains a node of which you wish to change the children, you can *reuse* that node using this type of rpattern. It is just like a "new" subtree, except that the label is replaced with a reference to the node in the original tree; this causes not only the label but also the coref index and all function tags to carry over to the new tree. The children, however, are replaced with those provided in the rpattern. Said children can themselves be "new", "reference", or "reuse" subtrees.

B.3.3 Replacing subpatterns

It sometimes happens to be easier to write an spattern where the base pattern is fairly high in the tree, but the node we wish to replace is further down, and therefore only a submatch of the spattern. To handle these cases, it is possible to specify *which* subpattern is being modified by the rpattern. To do so, simply precede the rpattern with a number (no backslash!) and a space. If this is not done, zero is assumed, which corresponds to the node matched by the entire spattern.

B.3.4 Some notes on the inner workings

In many respects, `tsed` acts very much like a combination of the `tgrep` tree-search utility and the `sed` text-replace (and more) utility. However, there is one important difference: when it handles references, it is actually handling them *by reference*. This is almost always what you would want, but may cause confusion in some cases. One consequent is that portions of trees cannot be copied—references can be used at most once each in the rpattern. If they are used more, the resulting behaviour is undefined.

Another, most noticeable when replacing subpatterns (cf Section B.3.3), is that nodes are removed before being reinserted. Thus, with an spattern like

```
A < [B] < [C < [D]]
```

(which would match a tree like (A B (C D E))), the rpattern

```
2 (\2 \1 \3)
```

would yield an output tree (A (C B D)). Note that although the initial 2 indicates that only the second subpattern (here labelled C) will be replaced, the first subpattern (B) is first removed from its place in the tree before being added back in as a child of C. (Also note that, due to the way the query was constructed, the E node just went away. That's a problem, we're working on it.)

B.3.5 Example rpatterns

Retagging

```
- LOC
```

This pattern keeps the structure of the input tree exactly the same, except that the matched node is tagged LOC. If the spattern had matched (PP ...) the result would be (PP-LOC ...).

Relabelling

```
RB
```

This pattern will change the label of the matched tree to RB. If the spattern matched (IN down), the output would be (RB down).

Adding structure

```
1 (S \1 \2)
```

This rpattern assumes there were (at least) two square-bracketted submatches in the affiliated spattern; it pulls them out of their respective places in the tree and puts them under a newly-created S node, which in turn is placed where the first of the submatches originally was. Thus if the spattern had been

```
A < [B] < [C]
```

which had matched (A (B ...) (C ...) ...), the output tree would be (A (S (B ...) (C ...)) ...).

B.4 Modifying the Penn treebank

The purpose for which `tsed` was written, and for which we envision it being used most often, was to provide a concise format with which to transmit modifications to large corpora—in particular, the Penn treebank. By itself, `tsed` can perform the modifications, but we need something a little more high-level to make modification scripts, and to handle the task of working specifically with treebank files. That program is `wsjsed`.

A script for `wsjsed` has, aside from blank lines and comments (lines whose first non-whitespace character is '`#`'), some number of `wsjsed` commands. Each of these is a directive to make a change to one sentence, one subsection, or the entirety of the Penn treebank, using calls to `tsed`. A command to `wsjsed` consists of three parts—the last two are the spattern and the rpattern, as described above.

The first part of the `wsjsed` command indicates which file(s) are to be modified, which sentences within those files, and any parameters to be passed to `tsed`, such as `-C` for case insensitivity. The exact format is

```
<subsection><file>#<sentence>[-<options>]
```

If there are no command line options, that portion can be omitted. To apply a change to all sentences in a given subsection, use the form

```
<subsection>*[-<options>]
```

To apply a change to the entire treebank, use

```
*[-<options>]
```

Note that the subsection and file number should always be two digits, even when less than 10 (pad with an extra zero if necessary). The sentence number can be any length.

B.4.1 Comments

Outside of a pattern, the hash character (‘#’) marks the start of a comment; everything else on that line will be ignored. In the usual case, this means that a line that starts with ‘#’ will be treated as blank, and ignored.

B.4.2 One-line commands

A one-line command has, predictably, all three parts on one line, each enclosed by curly brackets. The parts may be separated by whitespace. For instance, the line

```
{2421#5-C} {NP / stock} {VP}
```

in a `wsjsed` script would modify sentence 5 in file 21 of subsection 24 by making the NP headed by the word ‘stock’ (or ‘Stock’, or ‘STOCK’) into a VP.

B.4.3 Batch commands

If multiple modifications are to be made to the same file, it is more efficient to put them into the same `wsjsed` command, using the batch syntax. To invoke the batch syntax, you need to provide the `-b` option. If this option is given, there should be an open curly bracket (‘{’) on the line, and nothing else (aside from whitespace). Subsequent lines are treated as part of the batch, and will be given verbatim to `tsed` (see Section B.1); the batch is terminated by a line whose only non-whitespace character is a close curly bracket (}).[17]

```
{2415#19-b} {
  {IN < in}        {RP}
  {ADVP / in}      {! DIR}
  {ADVP / in}      {PRT}
}
```

will modify sentence 19 of file 15 of subsection 24 of the treebank, by retagging the word ‘in’ as a particle, removing the DIR tag from the phrase headed by it, and relabel that phrase as a particle phrase.

B.5 Installation

This is the subsection that needs the most work, because right now it is *extremely* ad hoc. There are many things that are ugly, requiring command-line configuration, that I’d eventually like to wrap into a relatively automatic “make install”-type script. For now, though, this is what there is.

[17] Yet another thing to be fixed eventually: make this a little less fragile with regards to what needs to be on which line.

B.5.1 Executables

If you have one of the systems that I claim to be 'supported', you don't need to download the source. There are three executables: `tgrep`, `tsed`, and `wsjsed`. For the moment, in order for them to work, you need to also grab the data directory containing files like "`headInfo.txt`". This, again, is something that will be fixed up in future releases, but for now you need to get this directory and put it somewhere. Whenever you run any of the executables, you will need to provide that directory's location with the `-D` option. Thus

```
wsjsed -D/home/myname/tseddata/ scriptfile
```

is what you would need to invoke, assuming that's where you put the data directory.

You may need to edit the first line of `wsjsed` if your system's perl executable is someplace other than `/usr/local/bin`.

The other (minor) issue is, of course, where to put the executables themselves. It shouldn't matter, although you'll probably want to make sure they are in your PATH somewhere.

B.5.2 Source

Here's where things get really hairy. I haven't been able to test this configuration on too many systems, so if you have problems, email me and we'll together work out what needs to be done.

You will need an ANSI-compliant C++ compiler (GNU's `g++` works admirably), and you will need `flex++` and `bison++`. The former comes installed on many systems, but you'll probably need to download the latter from the web—the sunsite mirrors have it. Finally, you will need to have `gmake` installed, although the primary make program (i.e. the one called when you simply type `make`) might be something else.

First, download and open the tarball. It will untar into a directory named `tsed`. That directory has a number of subdirectories; `data` is the directory mentioned above, that you'll need to provide `tsed` and the others with its path. `src` contains the source, obviously.

With any luck, you shouldn't need to play with anything in the subdirectories. The makefiles are set up in a pretty modular way, so that *hopefully* you will only need to create or modify the system-specific one. First, check what your ARCH environment variable is set to.[18] Edit the Makefile whose extension matches your ARCH: i686 for linux, etc. Probably the only thing you'll need to do is modify the locations of the various programs (`g++` et al).

Then, type '`make all`'. If it succeeds, `tgrep` and `tsed` will now be in the directory `$ARCH/O` (i.e. `ppc/O`, etc). `wsjsed` is in the main directory. Copy these three executables into some directory in your PATH.

[18] Check by typing '`echo $ARCH`' into a shell. If it's not defined, you'll probably want to set it. The ones we have predefined are 'sun4' (for Sun/Solaris), 'i686' (for Linux), and 'ppc' (for Mac OS X). Yes, we know that this should be an OS variable. No, we're not going to fix it right now.

If any of this doesn't work, email me—dpb@cs.brown.edu—and hopefully we can figure it out.

C Corrections applied to the treebank

These corrections are also available on the web at http://www.cs.brown.edu/~dpb/tbfix/.

Note that the misparse fix of 2428#2 was wrapped to fit the page, and should properly be on a single line.

```
#Type A
{24*-bg} {
  {NP !- LGS > (PP - LGS)} {- LGS}
  {PP - LGS} {! LGS}
}

#Type B
{2400#2}{NP < (NNP < York)}{- LOC}
{2400#2}{IN < about}{RB}
{2400#4}{IN < down}{RB}
{2400#6}{NP < (NNP < Wednesday)}{- TMP}
{2400#6}{ADVP / Meanwhile}{- TMP}
{2400#7}{PP / at << MMS}{- LOC}
{2400#11}{PP / in << imports}{- LOC}
{2400#14}{PP / in << prices}{- LOC}
{2400#14}{PP / in << consumer}{- LOC}
{2400#15}{PP / in << CPI}{- LOC}
{2400#16}{PP / among}{- LOC}
{2400#16}{PP / in << producer}{- LOC}
{2400#18-b} {
  {NP < (RB / as $. (_ / much) $.. (IN / as))} {QP}
  {NP / %} {- EXT}
}
{2400#20}{PP / in << inflation}{- LOC}
{2400#22}{PP / in << permits}{- LOC}
{2400#22}{PP / in << starts}{- LOC}
{2401#4}{IN < down}{RB}
{2402#2}{JJ < long}{VB}
{2402#7}{PP / in << detail}{- MNR}
{2402#8}{PP / in << room}{- LOC}
{2402#9}{IN < about}{RB}
{2402#14}{NP / morning}{- TMP}
{2402#21}{PP / on << globe}{- LOC}
{2402#23}{ADVP / still}{- TMP}
{2402#32-b} {
  {VP / make < [NP / him] < [ADJP / palatable]} {1 (S \1 \2)}
```

```
  {NP / him} {- SBJ}
  {ADJP / palatable} {- PRD}
}
{2402#34}{IN < about}{RB}
{2402#48}{PP / in << diaper}{- LOC}
{2403#2}{PP / to << Axa}{- DTV}
{2403#4}{IN < about}{RB}
{2403#12}{PP / in << states}{- LOC}
{2404#0}{IN < down}{RB}
{2404#1}{ADVP < NP < RBR}{- TMP}
{2404#2}{IN < down}{RB}
{2404#7}{IN < up}{RB}
{2404#12}{IN < down}{RB}
{2404#16}{PP / in << September}{- TMP}
{2404#27}{NP << between}{- EXT}
{2406#8}{SBAR <<, when << broadcasting}{- TMP}
{2406#32}{PP / Through}{- TMP}
{2406#36}{IN < around}{RB}
{2406#42}{NP / Newsreel}{- TTL}
{2407#5}{SBAR <<, because}{- PRP}
{2407#9}{PP / in << 1970s}{- TMP}
{2407#26}{NN < House}{NNP}
{2407#31}{ADVP / ever}{- TMP}
{2407#47}{PP / at}{- LOC}
{2409#0-g}{IN < about}{RB}
{2410#0}{PP / to << %}{- DIR}
{2410#2}{NP / quarter}{- TMP}
{2410#3}{IN < down}{RB}
{2411#3}{IN < about}{RB}
{2412#3-b} {
  {NP < [_ < Sen.] < [_ < Sam] < [_ < Nunn] < PRN} {1 (NP \1 \2 \3)}
  {NP / Ga} {- LOC}
  {NP / Okla.} {- LOC}
}
{2412#9}{PP / to << society}{- DTV}
{2412#33}{PP / about}{! LOC}
{2412#35}{PP / in << abstract}{! LOC}
{2412#50}{PP / in << U.S.}{- LOC}
{2412#69}{NP / behaviors}{- LGS}
{2412#69}{IN < down}{RP}
{2412#79}{SBAR <<, what}{- NOM}
{2413#9}{S / necessary}{FRAG}
{2413#21}{ADVP / ago}{- TMP}
{2413#23}{NP <<, Sunday}{- TMP}
{2413#27-b} {
  {IN < around} {RP}
```

```
  {ADVP / around} {! DIR}
  {ADVP / around} {PRT}
}
{2413#29}{S / to}{- PRP}
{2413#41}{ADVP / then}{- TMP}
{2415#0-b} {
  {IN < down} {RP}
  {ADVP / down} {! PUT}
  {ADVP / down} {PRT}
}
{2415#3}{PP / after}{- TMP}
{2415#4}{SBAR <<, what}{- NOM}
{2415#10}{NP / recovery}{- LGS}
{2415#17}{IN < about}{RB}
{2415#19-b} {
  {IN < in} {RP}
  {ADVP / in} {! DIR}
  {ADVP / in} {PRT}
}
{2415#45}{PP / in << account}{- LOC}
{2415#46}{IN < around}{RB}
{2416#7-b} {
  {PP / in << ownership} {- LOC}
  {NP / ownership} {! LOC}
}
{2417#4}{PP / in << volume}{- LOC}
{2417#15-b} {
  {VP / help < [NP / officials] < [VP / resolve]} {1 (S \1 \2)}
  {NP / officials} {- SBJ}
}
{2417#17}{SBAR <<, what}{- NOM}
{2417#20}{PP / After}{- TMP}
{2417#20}{IN < about}{RB}
{2417#22-b} {
  {IN < around} {RP}
  {ADVP / around} {! CLR}
  {ADVP / around} {! LOC}
  {ADVP / around} {PRT}
}
{2417#33-b} {
  {VP / having < [NP / trades] < [VP / flow]} {1 (S \1 \2)}
  {NP / trades} {- SBJ}
}
{2417#41}{NP / points}{- EXT}
{2417#45}{IN < about}{RB}
{2417#51}{PP / in << futures}{- LOC}
```

```
{2417#56}{NP / proposals < PRN}{- HLN}
{2417#66}{IN < about}{RB}
{2417#70}{IN < about}{RB}
{2417#71}{PP / at}{- LOC}
#{2417#73}{VBD < bribed}{VBN}
{2417#75}{IN < about}{RB}
{2417#84}{IN < about}{RB}
{2417#87}{IN < down}{RB}
{2417#87-b} {
  {SBAR < [S < (NP < [DT < that])]} {2 IN}
  {SBAR < [S < (NP < [IN < that])]} {(SBAR \2 \1)}
}
{2417#88}{DT < half}{NN}
{2417#88-b} {
  {NP > (PP / in << half)} {! TMP}
  {PP / in << half} {- TMP}
}
{2417#91}{IN < about}{RB}
{2418#12-g}{IN < about}{RB}
{2418#25}{ADJP / necessary}{- PRD}
{2418#25}{IN < on}{RB}
{2418#30}{ADVP / initially}{- TMP}
{2418#30}{IN < about}{RB}
{2418#34}{PP / In << papers}{- LOC}
{2418#37}{ADVP / so}{- PRD}
{2418#45}{PP / with << suit}{- MNR}
{2418#49}{PP / in << case}{- LOC}
{2419#2}{NP <<, Exeter << N.H.}{- LOC}
{2419#2}{NP <<, Fitchburg << Mass.}{- LOC}
{2422#0}{NP <<, New << York}{- LOC}
{2422#2}{NP <<, Tampa << Fla.}{- LOC}
{2422#2}{PP / for << year}{- TMP}
{2422#3}{S / is <<‘ basis}{- TPC}
{2424#3}{ADVP / ever}{- TMP}
{2424#3}{IN < about}{RB}
{2425#0}{IN < about}{RB}
{2426#6}{SBAR <<, which}{- NOM}
{2427#2}{PP / after << voting}{- TMP}
{2428#2}{VP < [VB < mark] < (S < [_ / down] < [NP / quotations])
    < [SBAR <<, while]}{(VP \1 \2 \3 \4)}
{2428#2-b} {
  {IN < down} {RP}
  {ADJP / down} {! PRD}
  {ADJP / down} {PRT}
}
{2428#4-b} {
```

```
  {S} {! TPC}
  {S} {! 1}
  {S < [NP / calamity] < [VP / is] < .} {1 (S \1 \2)}
  {S <<` over} {- TPC}
  {S <<` over} {- 1}
}
{2428#8}{NP < (CD < 2002)}{- TMP}
{2428#21}{IN < about}{RB}
{2428#23}{IN < about}{RB}
{2428#23-b} {
  {PP / to << 7.16} {! EXT}
  {PP / to << 7.16} {- DIR}
}
{2428#25}{SBAR <<, When}{- TMP}
{2428#46-b} {
  {(NP - ADV < (_ < iota))} {- EXT}
  {(NP - ADV < (_ < iota))} {! ADV}
}
{2428#49}{NP <<, Olympia}{- LGS}
{2428#51}{PP / before << Campeau}{- TMP}
{2428#52}{IN < down}{RB}
{2428#53}{NP / months}{- TMP}
{2428#54}{IN < about}{RB}
{2428#62-b} {
  {NP < [SBAR]} {\1}
  {SBAR <<, What} {- NOM}
  {SBAR <<, What} {- SBJ}
}
{2428#64-b} {
  {NP < [SBAR]} {\1}
  {SBAR <<, What} {- NOM}
  {SBAR <<, What} {- SBJ}
}
{2428#67}{NP / Thursday}{- TMP}
{2428#68}{PP / since << March}{- TMP}
{2428#75}{NP <<, about <<` %}{! EXT}
{2428#75}{IN < about}{RB}
{2428#76}{IN < about}{RB}
{2428#76}{IN < up}{RB}
{2428#81-b} {
  {NP < (NP < [RB < as] < [RB < much]) < (PP < [IN < as] < (NP < [CD] < [NN]))}
   {(\0 (QP \1 \2 \3 \4) \5)}
  {NP < QP} {- EXT}
}
{2429#0-b} {
  {VP / keep < [NP] < [ADJP]} {1 (S \1 \2)}
```

```
  {NP <<, markets} {- SBJ}
}
{2429#1}{IN < about}{RB}
{2429#2}{ADVP / immediately}{! TMP}
{2429#4}{ADVP / shortly}{! TMP}
{2429#23}{PP / under <<` pressure}{! LOC}
{2429#24}{PP / in <<` 1987}{- TMP}
{2431#0}{NP <<, this / year}{- TMP}
{2431#17}{ADVP / regularly}{- TMP}
{2431#26}{NP << coalition / birth}{- LGS}
{2431#29}{PP / in << shuffle}{- LOC}
{2431#29-b} {
  {PP / for << Greece} {! PRD}
  {ADJP < (JJ < Crucial)} {- PRD}
  {ADJP < (ADJP / Crucial)} {- ADV}
  {VP < [_ < are] < [PP / for]} {(\0 \1 (ADJP (-NONE- *?*)) \2)}
  {ADJP < -NONE-} {- PRD}
}
{2432#0}{PP <<, following}{- TMP}
{2432#6-b} {
  {IN < around} {RP}
  {ADVP / around} {! DIR}
  {ADVP / around} {PRT}
}
{2432#16}{IN < down}{RB}
{2432#16}{VBD < dragged}{VBN}
{2433#17}{PP / to << defunct}{- DTV}
{2433#23}{IN < about}{RB}
{2433#24}{IN < about}{RB}
{2433#36}{NP < [_ < about] < [_ < eight] < (_ < months)}{1 (QP \1 \2)}
{2433#36}{IN < about}{RB}
{2435#7}{IN < around}{RB}
{2436#5}{NP / today}{- TMP}
{2437#2}{IN < about}{RB}
{2438#0-g}{IN < about}{RB}
{2438#1-g}{IN < about}{RB}
{2438#3}{IN < about}{RB}
{2438#4}{PP / in << face}{! TMP}
{2438#6-g}{IN < about}{RB}
{2438#7-g}{IN < about}{RB}
{2438#8}{IN < about}{RB}
{2438#10-g}{IN < about}{RB}
{2438#11}{IN < about}{RB}
{2438#14-g}{IN < about}{RB}
{2440#3-g}{IN < about}{RB}
{2442#4}{NP <<, a / share}{- ADV}
```

```
{2442#4}{IN < about}{RB}
{2443#7}{SBAR <<, where}{- NOM}
{2443#28}{IN < about}{RB}
{2443#43-g}{IN}{RB}
{2443#50}{IN < down}{RB}
{2443#55}{PP / on << loans}{! LOC}
{2443#59}{PP / among}{- LOC}
{2444#11}{IN < down}{RB}
{2444#14-b} {
  {SBAR <<, Once} {! ADV}
  {SBAR <<, Once} {- TMP}
}
{2444#15}{NP << staggering / %}{- EXT}
{2444#20}{PP / in << years}{- TMP}
{2444#21}{IN < down}{RB}
{2444#21-b} {
  {PP / from << peak} {! TMP}
  {PP / from << peak} {- DIR}
}
{2444#37}{PP / in <<` 1983}{- TMP}
{2444#38}{NP / year <<` available}{- TMP}
{2444#41}{IN < down}{RB}
{2444#41-b} {
  {PP / from <<` peaks} {! TMP}
  {PP / from <<` peaks} {- DIR}
}
{2444#45}{PP <<, even / in}{- LOC}
{2444#45}{VP < (RB < even)}{ADVP}
{2444#48}{ADVP / ahead}{- TMP}
{2446#6}{PP / with <<` money}{- MNR}
{2446#15}{NP < [_ < about] < [_ < two] < (_ < weeks)}{1 (QP \1 \2)}
{2446#15}{IN < about}{RB}
{2446#16}{S / defrauding <<` Lincoln}{- NOM}
{2446#34}{SBAR <<, what << investigated}{- NOM}
{2448#1}{IN < about}{RB}
{2448#3}{PP / in << wake}{- LOC}
{2448#16-b} {
  {SBAR / while <<` billion} {! TMP}
  {SBAR / while <<` billion} {- ADV}
}
{2448#30}{IN < about}{RB}
{2448#35}{IN < down}{RB}
{2448#40}{PP / in <<` Guangdong}{- LOC}
{2449#3-b} {
  {PP / in << brief} {- LOC}
  {NP / brief  < VP} {! LOC}
```

```
}
{2450#0}{VBD < hit}{VBN}
{2450#3}{IN < around}{RB}
{2450#9}{ADVP / usually}{- TMP}
{2450#10}{IN < up}{RB}
{2450#12}{IN < down}{RB}
{2451#13}{SBAR <<, how}{- NOM}
{2451#13-b} {
  {IN < down} {RP}
  {ADVP / down} {PRT}
  {PRT - DIR} {! DIR}
}
{2451#15}{S > SINV}{- TPC}
{2451#15}{SBAR <<, what}{- NOM}
{2451#19}{PP / on << side}{- LOC}
{2451#26-b} {
  {JJ < unshackled} {VBN}
  {ADJP - PRD < [VBN] $ [S]} {(VP \1 \2)}
  {VP - PRD} {! PRD}
}
{2451#29}{NP / enterprises}{- LGS}
{2451#39}{PP / in <<` Washington}{- LOC}
{2453#1}{IN < down}{RB}
{2453#4}{IN < down}{RB}
{2453#6}{IN < down}{RB}
{2453#11}{IN < down}{RB}
{2454#9}{PP / in <<` Zambia}{- LOC}
{2454#11}{NN < back}{RB}
{2454#14}{NP / night}{- TMP}
{2454#20-b} {
  {SBAR <<, As} {! TMP}
  {SBAR <<, As} {- ADV}
}
{2454#24}{PP / under <<` pressure}{! LOC}
{2454#25}{S < (VP <<, to << see)}{- PRP}
{2454#25-b} {
  {SBAR <<, if} {! ADV}
  {SBAR <<, if} {- NOM}
}
{2454#29-b} {
  {VP / see < [S]} {1 NP}
  {NP - SBJ / lions} {! SBJ}
  {PP - PRD / in <<` action} {! PRD}
}
{2454#32}{PP / on <<` followers}{! LOC}
{2454#34}{ADVP / out}{PP}
```